Confidence in Dynamic Assurance Cases

by

Simon Diemert
B.S.Eng., University of Victoria, 2015
M.Sc., University of Victoria, 2017

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

We acknowledge and respect the Lək̓ʷəŋən (Songhees and Xʷsepsəm/Esquimalt) Peoples on whose territory the university stands, and the Lək̓ʷəŋən and W̱SÁNEĆ Peoples whose historical relationships with the land continue to this day.

Confidence in Dynamic Assurance Cases

by

Simon Diemert
B.S.Eng., University of Victoria, 2015
M.Sc., University of Victoria, 2017

Supervisory Committee

Dr. Jens H. Weber, Supervisor
(Department of Computer Science)

Dr. Hausi Müller, Departmental Member
(Department of Computer Science)

Dr. Issa Traoré, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Assuring safety- and security-critical systems is a necessary activity, both prior to deployment (at "design-time") and during system operation (at "run-time"). The need for assurance is heightened as these systems increasingly depend on artificial intelligence and adaptation strategies to handle uncertainty in their operating environments. Assurance Cases (ACs) that incorporate structured arguments and supporting evidence are an important tool for establishing trust in critical systems. Modern ACs are not static documents: they are increasingly viewed as dynamic models of "through-life" assurance and are used for decision-making at both design- and run-time. These Dynamic Assurance Cases (DACs) incorporate dynamic sources of evidence and "live" data from development activities or operations (e.g., safety performance indicators). However, a question arises during their use: "*are we confident in the claim(s) made by this version of the case?*"

While several methods exist to assess confidence in ACs, there is limited knowledge about their adoption by practitioners. Additionally, there are several limitations of quantitative methods, including: 1) an inability to consider the impact of dynamic aspects on confidence; 2) an inability to account for dialectic reasoning (i.e., "defeaters"); and 3) challenges related to subjectivity, interpretability, precision, and modelling nuanced reasoning. The overall objective of this work is to develop a new confidence assessment method that is grounded in the needs of practitioners and addresses the limitations mentioned above.

Towards this objective, the main contribution of this dissertation is a new mixed (qualitative and quantitative) method for AC confidence assessment called *Certus*. With this method, confidence in an AC is modelled using vague, but linguistically meaningful, expressions (e.g., "I have very high belief that this claim is true"). A domain specific language is used to describe the propagation of belief through a structured argument to produce an overall belief valuation for the AC. *Certus* supports dialectic reasoning and can condition belief assessments on the availability of evidence and the value of performance indicators. The use of the language is guided by a methodology that integrates with the existing practices for developing (D)ACs. A denotational semantics for the language provides a formal basis for assessment. The language and method are evaluated through a series of analyses and a case study to demonstrate that they possess properties related to trustworthiness, including: propagation stability, sensitivity, expressivity, scalability, and applicability to DACs.

Table of Contents

List of Tables

List of Figures

Acknowledgements

The work in this dissertation did not occur in a vacuum. I have been fortunate to have many collaborators, colleagues, and mentors who have influenced this dissertation, related projects, and supported my growth as an engineer and researcher.

Thank you to the many collaborators who contributed to projects and papers in recent years: Caleb Shortt for contributing as a co-investigator to our interview study of practitioners; to Prof. Marsha Chechik, Dr. Torin Viger, and colleagues from the University of Toronto for our collaboration on safety assurance and generative AI; and Dr. Chuck Weinstock and Dr. John Goodenough from the Software Engineering Institute for their encouragement, particularly early in my research.

Thank you to the University of Victoria and the Natural Science and Engineering Research Council (NSERC) of Canada. Both of these organizations have provided financial support for this in the form of grants and scholarships.

Thank you to the members of my supervisory committee, Prof. Hausi Müller and Prof. Issa Traoré, for their guidance over the last four years. On several occasions, their suggestions influenced the direction of my research, ultimately resulting in a stronger result. Additionally, I would like to thank Prof. Alan Wassyng for acting as the external examiner for this dissertation, and for his sincere and thoughtful engagement with these ideas during the oral exam.

Thank you to my colleagues at Critical Systems Labs (CSL) for their expertise, support, and friendship. My experience to date with CSL has been a tremendous learning experience that has developed my perspective on engineering, business, systems and software assurance, research, and leadership. My practical experiences from CSL have provided invaluable context that I believe improved the depth and applicability of the results in this dissertation. I would especially like to thank Dr. Laure Millet for her willingness to go along with my "crazy" ideas (or tell me I'm wrong), Lorie Joyce for her trust and support, and Dr. Jeff Joyce, whose lessons, mentorship, and wisdom I will undoubtedly return to for many years to come.

To my supervisor, Prof. Jens Weber, I would like to express a deep gratitude for your long-term support, knowledge, and guidance. I had no idea, when I walked into your office in 2013 as an undergraduate student that it would culminate in me completing a Ph.D.! Thank you for believing in me and helping me see that this was indeed possible.

Thank you to my friends and family for your unconditional support and engagement with my work, especially to those who have read some of the papers and drafts of this dissertation, and asked questions.

Finally, to India Wiebe, my partner. Thank you for your understanding, patience, listening to my many worries, doubts, and half-baked ideas. And most importantly, for being present every day.

Chapter 1
Introduction

On the evening of March 18th, 2019, a 49-year-old woman named Elaine Herzberg was struck and killed by an autonomous vehicle operating in "test mode" as she was crossing the road with her bicycle outside of Tempe, Arizona [1]. The vehicle's perception system detected an object on the road several seconds before the crash, but it did not correctly classify the object as a pedestrian and failed to track Ms. Herzberg as she moved across the road [1, p. 39]. Though a safety driver was present to supervise the automation, they did not respond in time to prevent a collision. Ms. Herzberg's death is thought to be the first fatality involving an autonomous vehicle. Subsequently, the rapid deployment of autonomous and advanced assistive driving technologies has led to several accidents, including some resulting in severe injuries and fatalities [2, 3].

Autonomous vehicles are one example of a *software-intensive critical system*. Such systems depend on software to realize security-, mission-, or safety-critical capabilities. Modern society depends on software-intensive critical systems in a range of application areas, including transportation, energy, manufacturing, defence, finance, and health. While these systems provide great benefit, they also introduce risk as evidenced by many high-profile accidents [4, 5, 6, 7, 8].

Definition 1 (Software Intensive System) *"Any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole" [9].*

Definition 2 (Critical System) *A system that satisfies at least one of the following: 1) the incorrect, inadvertent, out-of-sequence, or failed response of the system might contribute to a loss event; 2) the system is intended to mitigate a loss event; or 3) the system is intended to recover from a loss event [9, 10].*

In the above definition of a *critical system*, the term *loss event* refers to many types of situations involving some manner of loss or harm [11]. The specific definition depends on the quality attribute(s) of interest: in a safety-critical context the term *accident* is used to refer to loss events, possibly resulting in severe injury or loss of life; and for a security-critical information system a loss event might be disclosure of sensitive data. In a safety-critical context, losses are preceded by system conditions called *hazards* [10, 11]. If a hazard occurs under the right conditions, and is not mitigated, then it can precipitate into an accident. Significant engineering effort, including the use of methods described in this dissertation, is spent to identify, prevent, or mitigate the occurrence of hazards. Similar terminology exists for other quality attributes; for instance, in a security context these conditions are called *threats* [12].

Many modern systems are now using Artificial Intelligence (AI) to implement functionality that was extremely difficult (or impossible) to achieve with conventional software. In some cases, AI is used as part of critical control functions. We refer to these as *AI-enabled systems*. For instance, perception systems for autonomous vehicles use complex machine-learnt statistical models to detect and classify objects near the vehicle. While AI enables systems to generalize to novel situations, it raises challenges for managing the already significant risk associated with a critical system's deployment. This dissertation is being written at a time when a growing number of organizations are incorporating AI into safety-related functions within critical systems, for example, using AI as part of safety-related control paths or protection systems intended to detect and respond to hazards. Consequently, there is an urgent need to advance methods that assist engineers in assuring AI-enabled systems.

1.1 Assuring Modern Critical Systems

Engineering process is a cornerstone of assurance for critical systems. Many industrial standards, regulatory frameworks, and guidelines describe activities that organizations must undertake as part of system and software engineering development lifecycle(s). As the risk associated with a system increases, more demanding activities are required to demonstrate that the risk is adequately mitigated. For instance, in the automotive industry ISO 26262 identifies formal verification as an "optional" activity for lower criticality software (ASIL A and B) but "recommends" it for higher criticality software (ASIL C and D) [13]. This strategy of increasing rigour to address increased risk is called the *level of rigour* approach and is intended to reduce the

likelihood of latent systematic defects impacting the operation of a system.

However, the process-oriented level of rigour approach does not prescribe specific properties that a software-intensive system must satisfy. For instance, ISO 26262 does not limit the cyclomatic complexity of safety-critical software functions or modules used in automotive systems; it only "strongly recommends" that programming languages and guidelines be chosen to enforce low complexity software [13, Part 6, p. 7]; and similar approaches are taken for airborne software [14] and rail signalling software [15]. An organization can develop software-intensive critical systems that are compliant with industrial standards without necessarily demonstrating that they satisfy certain properties. That is, engineering process rigour is used as a proxy for dependability of software-intensive systems.

For complex critical systems, especially for those that cede control to AI, applying a suitably rigorous engineering process is necessary, but not sufficient, for assurance [16, 17]. It is also necessary to provide a detailed rationale for how critical quality attributes such as safety or security are achieved. This is especially true for AI-enabled systems where the lifecycle used to develop AI differs significantly from established software engineering processes [18, 19]. In short, something more than engineering process is required for assuring a complex critical system, like an autonomous vehicle.

1.2 Assurance Cases

In addition to following a rigorous engineering process, it is now common to prepare an *assurance case* that argues, with evidence, that a critical system has achieved its desired quality attribute(s):

Definition 3 (Assurance Case (AC)) *A reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will satisfy its desired quality attributes when operating on a defined application in a defined environment* (adapted from [20]).

The term "assurance case" is a general term used to refer to a wide range of quality attributes. By convention, the name identifies the quality attribute(s) of interest, e.g., "safety case" or "security case". The contributions in this dissertation are broadly applicable to ACs expressed using the structured notations discussed below, though many examples select safety as the quality attribute of interest.

Preparing and maintaining an assurance case is widely recognized by researchers

[19, 21], industrial associations [20, 22], regulators [10, 12], companies [23, 24], and technical standards bodies [25, 26] as necessary for complex systems, including those that depend on AI.

1.2.1 Expressing Assurance Cases

The modern notation for expressing an AC's argument was introduced by Kelly [27] and concurrently by Bishop and Bloomfield [28] in response to a series of major accidents, including the death of 167 workers in the explosion and collapse of the Piper Alpha oil drilling platform in 1988 [29]. At the time, there was a call for system designers and operators to produce a safety argument "demonstrat[ing] a systematic and thorough consideration of safety" [27].

Building on the foundation for structured arguments from philosophy by Toulmin [30], the Goal Structuring Notation (GSN) [27] and Claims-Argument-Evidence (CAE) notation [28] were created to express goal-oriented assurance arguments in a uniform and structured manner. Other notations for representing assurance arguments have since been proposed, including Eliminative Argumentation (EA) and the Friendly Argument Notation (FAN) [31, 32]. Compared to an assurance report (resp. "safety report") written in a narrative style, an AC expressed in a notation like GSN helps to organize the argument's logic and supporting evidence, improves readability, and provides a semi-formal model of an argument that can be used as the basis for specialized tool support and automation.

1.2.2 Industrial Adoption of Assurance Cases

Industrial adoption of ACs varies by industry and jurisdiction. In some cases, ACs are mandated by laws and regulations, while in others ACs must be prepared to achieve compliance with industrial standards but are not a legal or regulatory requirement. As evidenced by the survey of industrial adoption below, ACs are widely recognized as an important artifact for critical (especially safety-critical) systems. Moreover, there is a trend towards increasing adoption of ACs as systems become more complex and incorporate AI-enabled capabilities.

Automotive. In the automotive industry, organizations seeking compliance with standards such as ISO 26262 (functional safety) [13], ISO 21434 (cybersecurity) [33], UL 4600 (autonomy) [34], and ISO/PAS 8800 (AI-enabled vehicles) [25] must prepare

an AC for safety- or security-critical electronic and software systems. At present, in Canada and the USA, compliance with these technical standards is not required by law, but is demanded by market forces in the automotive supply chain. Further, the United Nations is preparing a Global Technical Regulation on the safety of automated driving that, once adopted by member states, will require the preparation of a safety case for automated vehicle technology [35].

Nuclear Power. In the nuclear industry in Canada a safety case must be submitted to (and accepted by) the Canadian Nuclear Safety Commission to receive a license to operate a nuclear facility or manage nuclear waste [36]. Similar requirements exist in the United Kingdom [37]. Additionally, there is growing interest in the use of structured argumentation in this domain in the USA [38].

Rail. In the rail industry, organizations seeking compliance with EN 50126 for rail signalling systems must prepare a safety case report [39].

Medical Devices. In the medical device industry, there is interest in ACs. The US FDA has acknowledged AAMI's TIR38 guidance on assurance case reports as a requirement for infusion pumps [40]. However, at least in Canada and the USA, there does not appear to be regulatory requirements or standards that demand the preparation of an AC for a generic medical device.

Commercial Aviation. ACs for software and electronics are not typically prepared for commercial aviation systems which have historically depended upon very rigorous engineering processes (e.g., DO-178C for airborne software) for assurance. However, there is interest in preparing ACs to argue that "overarching properties" are satisfied for this type of system [41].

Defence. Some countries, such as the United Kingdom, require that organizations delivering defence technology prepare an AC [42]. In Canada, the Royal Canadian Navy is preparing to publish a Canadian Forces Technical Order requiring that all vessels procured by the Canadian Department of National Defence prepare a safety case as part of their Naval Materiel Safety Management program [43].

Oil & Gas. In the United Kingdom there are statutory requirements in the oil and gas industry to prepare assurance cases for offshore extraction platforms and equipment [44].

Professional Engineering in B.C. In British Columbia, Canada, it is recommended by the engineering regulator (Engineers and Geoscientists of B.C.) that engineers developing safety- or security-critical software prepare an assurance case to satisfy their professional obligations [10, 12].

1.2.3 The Assurance Case Method

There is no single universally recognized AC method that is used to prepare (or maintain) ACs. In fact, a preliminary step when preparing an AC is to select a suitable combination of notations, (sub-)methods, and procedures that will be used. In some cases, aspects of methodology are pre-determined by industry-specific conventions or requirements, or organizations have processes that define *their* AC methodology. Proposals for end-to-end AC methodologies exist (e.g., [45, 46]) which have shared themes that point to core aspects of a methodology. Chapter 2 formulates an AC method as a means of establishing the foundation for the contributions of this dissertation.

1.3 Confidence Assessment for ACs

While there is strong (and still growing) adoption of ACs as a means to capture arguments and evidence, not all arguments and evidence are equal. Arguments that are convincing in one setting might not be as compelling in another, or evidence might have been produced through untrustworthy means. Structured methods for expressing ACs provide some rigour by establishing a consistent syntax and semantics, which can aid in analysis and evaluation. However, preparing an AC remains a subjective task that is sensitive to both the quality of the argument and the supporting evidence. An important question arises: *how can one be confident that the claim(s) made by an AC are true?*

Many researchers have proposed methods for assessing confidence in ACs [47, 48, 31, 49, 46, 50]. Collectively, this dissertation refers to these methods as *confidence assessment methods (CAMs)*. These methods can be organized into three categories: qualitative, quantitative, or mixed. Qualitative methods systematically apply procedures, criteria, or notations to reason about confidence in the argument [47, 49, 51]. Quantitative methods represent confidence (or related measures such as "belief") in the AC's top-level claim numerically. Mixed methods incorporate both qualitative and quantitative aspects [31, 46].

Definition 4 (Confidence Assessment Method (CAM)) *A method, procedure, or activity undertaken to determine if one or more claim(s) in an assurance case have adequate support such that they can be considered to be true for the purpose of assuring one or more quality attributes of a critical system.*

1.3.1 The Role of Confidence Assessment

As evidenced by the breadth of research in this area and the opinions of practitioners [52], confidence assessment is regarded as an important activity when preparing or maintaining an AC. Several aspects of the value proposition for confidence assessment are described below.

Value Proposition: Systematic Review and Interrogation

CAMs provide an opportunity to systematically review or interrogate an AC's argument and supporting evidence. This process can reveal deficiencies in an argument, the underlying system, or supporting processes and procedures. Several CAMs include elements of systematic review or interrogation. For instance, the EA method encourages AC developers to express and reason over "doubts" they might have [31], and Holloway and Wasson's iTest method includes criteria and questions that can be used to structure reviews of an AC [49]. For quantitative CAMs, the activity of assigning numerical confidence valuations (and other parameters) required to complete the calculations creates opportunities for users to engage in a process of reflection and critical thinking.

Value Proposition: Communication and Decision-Making

Since ACs capture the assurance rationale for a whole system or organization, they are valuable tools for communicating with interest holders and supporting them in making informed decisions about a system or organization (e.g., "are we confident that we can deploy, operate, and decommission the system safely?"). CAMs have the potential to further improve communication. For instance, CAMs can be used to highlight residual sources of doubt in an AC that may be brought to the attention of interest holders [53]. When quantitative CAMs are used, they can provide input to (semi-)automated decision processes, which enable advanced system capabilities, such as adaptation (discussed below).

Value Proposition: Mitigating Confirmation Bias

CAMs help to mitigate bias that might be introduced during the development of an AC [22]. Of particular importance is confirmation bias, wherein developers accept claims or evidence that confirms pre-conceived notions about the system. ACs have been criticized as being prone to confirmation bias [54], and confirmation bias is thought to have played a role in fatal accidents, such as the crash of the Nimrod aircraft in 2006 [55]. Dialectic reasoning is a strategy that helps to mitigate confirmation bias through the expression of doubt in the argument (i.e., "counter-claims" or "defeaters") [22, 53, 46].

Value Proposition: Through-Life Confidence Assessment

It is possible that confidence in an AC will change over time [56, 57, 51]. If an AC is created at the beginning of system development, then confidence in the AC should (ideally) increase as the design matures and verification and validation evidence is produced to demonstrate that the system will satisfy its assurance objectives [57]. When a system is deployed, confidence might decrease as the assumptions made in the AC are challenged during operation, or if the operational environment changes [56, 51]. Therefore, an important role of CAMs is to provide an objective means of evaluating confidence throughout the system's lifetime.

1.4 Dynamic ACs for 'Through-Life' Assurance

Despite early work from Kelly that positioned GSN as a means of supporting AC management throughout the lifetime of a system [27], ACs have largely been viewed as "one-off" artifacts that are prepared at the end of a project to wrap up assurance activities prior to a system's deployment. That is, an AC was viewed as a decision-making tool about whether the system could be deployed, but then not necessarily maintained after deployment. However, in the last decade, particularly as critical systems have become more sophisticated (e.g., using AI), there has been growing interest in *through-life assurance* [27, 56]. In the through-life perspective, AC development begins during system conceptual design [58] and continues through system operation [21, 34], with the AC being updated many times as the system evolves.

Two complementary concepts related to through-life assurance have emerged: *continuous assurance* and *perpetual assurance*. Both of these ideas view an AC as a

dynamic model of assurance that can be used to make decisions about whether (a version or configuration of) a critical system can be deployed. Each of these concepts is discussed in turn below, and then the notion of a *dynamic assurance case* is introduced, as it relates to continuous and perpetual assurance.

1.4.1 Continuous Assurance

Along with increases in system complexity, there is a desire to reduce development cycle time for critical systems by adopting iterative ("agile") development models and making use of principles from DevOps [59]. However, even with faster development cycles, it is still necessary to assure each change to a critical system. The model-based perspective offered by structured argumentation methods, like GSN, allows ACs to integrate with (semi-)automated development methodologies and tools, e.g., Continuous Integration and Continuous Delivery (CI/CD) pipelines [60, 61]. When applied to critical systems, this is referred to as *DevSafeOps* [61], or more generally *continuous assurance* [62].

As in the DevOps approach, the notion of continuous assurance extends beyond design-time development activities to include run-time monitoring of the deployed system [61]. Quality attribute-specific *measures* or *performance indicators* (e.g., Safety Performance Indicators (SPIs)) are used to monitor the behaviour of the deployed system [56, 34, 21], and can be used to instrument the AC to determine if claims made in the argument remain valid during system operation [56, 63, 64].

As a concrete example, Zeller et al. have developed the "MLOps" framework and applied it to an AI-enabled rail control system called safe.trAIn [65, 66]. In their approach, the AC is regularly updated as part of iterative system development activities. Verification and validation evidence produced by CI/CD pipelines is automatically incorporated into the AC so that it remains up to date with the latest evidence produced during system development. For their safe.trAIn application, SPIs are monitored and used to (in)validate claims in the AC about AI model transparency. In the MLOps framework, each version of the AC is subject to safety assessment by an independent authority with a focus on the most recent changes.

1.4.2 Perpetual Assurance in Self-Adaptive Systems

Assurance for most critical systems, especially those that are safety-critical, depends on the assumption that the system will only be used when the operational environ-

ment and system's internal conditions satisfy a set of pre-determined constraints. For example, autonomous vehicle companies operating fleets of vehicles will limit operations to "sunny day" conditions and halt operations if weather conditions deteriorate. While autonomous vehicles are already equipped to handle a significant level of uncertainty in their environment, their ability to handle truly novel situations is limited by what was anticipated by system designers. Like many types of critical systems, autonomous vehicles have their behaviour fixed at design-time, a fact that their AC arguments depend on. Consequently, these systems cannot adapt their behaviour to accommodate fundamentally different conditions.

Self-Adaptive Systems (SASs) are a category of systems that are able to adjust their behaviour at run-time to accommodate changes in their environment or internal changes (e.g., component failures) [67]. The architecture of a SAS usually consists of a *managed system* and a *managing system*. According to Weyns, the managed system "interacts with the environment and is responsible for domain concerns", and the managing system "consists of a feedback loop that interacts with the first and is responsible for the adaptation concerns" [67]. The managing system aims to satisfy adaptation goals (provided by system operators) by adjusting the behaviour of the managed system at run-time.

There is an interest in applying self-adaptation to critical systems to manage uncertainty at run-time. However, doing so requires methods for assuring SASs, which is an open research challenge [68, 67]. A key idea is *perpetual assurance* where new evidence is automatically generated (by both the managing system and humans, where appropriate) to ensure the SAS continues to address uncertainty in its operation [68]. Among the methods available to support perpetual assurance, ACs have been identified as a means of composing disparate pieces of evidence together towards an overall assurance goal [68]. It is possible to reason about the safety of a SAS's adaptation based on the impact the adaptation has on an AC [69, 70, 71]. Consequently, SASs can use ACs as a run-time model of assurance ("models @ run-time"). The model can be adapted at run-time to reflect the assurance rationale for new system configurations [72, 69, 73, 74]; and supporting evidence can be generated in the adaptation process, possibly by simulation with digital twins [75, 76, 77].

1.4.3 Dynamic Assurance Cases

The two concepts above have been developed in different fields. On one hand, *continuous assurance* emerged from the need to maintain assurance, even with shorter development cycle times. On the other hand, *perpetual assurance* originates from the self-adaptive systems' community with a focus on assuring systems as they adapt or evolve to manage uncertainty. From these two concepts, five shared themes emerge:

1. **Structured Arguments** - ACs are represented in a structured form, using notations like GSN, that allows them to be viewed as models of assurance [27, 28].

2. **Evolving Rationale** - AC arguments are used for through-life assurance, with the AC's model of assurance rationale evolving to reflect changes in the system or its operational environment.

3. **Updating Evidence** - ACs should be updated with the latest evidence, either generated by design-time activities, or produced as part of run-time adaptation activities [75].

4. **Indicators from Live Data** - ACs should be instrumented with indicators (e.g., SPIs) that draw on "live" data from development activities, system operations, or the environment to determine if their claims remain valid [56, 63].

5. **Decision-Making** - ACs are used for decision-making about whether to deploy a system or if an adaptation will achieve critical quality attributes.

Collectively, these themes triangulate the notion of a *Dynamic Assurance Case* (DAC), a term introduced by Denney et al. in 2015 [56] and further developed by Asaadi et al. [63]. However, Denney et al.'s original definition also adds the notion of confidence assessment as a basis for making decisions. Integrating these ideas, we adopt the following functional definition of a DAC.

Definition 5 (Dynamic Assurance Case (DAC)) *A through-life model of assurance rationale that evolves alongside the system, incorporating up-to-date reasoning, evidence, and measures, where the confidence that a system will satisfy critical quality attributes is (re-)assessed for every change made to the model and used as a basis for decisions about whether to deploy or (continue to) operate a version of a system.*

The above definition of a DAC is "functional" in the sense that it describes the purpose of a DAC as part of a larger systems development and evolution process. That is, a DAC is used as a decision-making tool based on available assurance rationale, evidence, and data. This definition is intended to complement other definitions of DACs that are compositional[1] in nature, focusing on core elements (argument, evidence, measures, confidence) of a DAC [56, 63]. Such elements are essential for implementing a DAC as contemplated by Definition 5.

1.5 Problem Formulation

This dissertation considers the problem of assessing confidence in DACs, as formulated above. Despite a history of industrial and academic research in this area, several problems and gaps in knowledge remain.

1.5.1 Missing Knowledge on the Adoption of CAMs

Though CAMs have been the subject of research for nearly two decades, there is limited empirical evidence about how practitioners approach confidence assessment for ACs. Nair et al.'s 2012 survey briefly explored this topic and reported that practitioners felt confidence assessment was an important or very important challenge [78]. However, no additional information was reported on how survey respondents approached confidence assessment in practice. Cyra and Gorski performed a survey to calibrate a qualitative scale used to provide inputs to their DST-based method, but the report on the survey method is limited in detail and, since it was focused on calibration, the results are very specific to their method [79]. The limited survey data is compounded with few published case studies or experience reports describing the use of CAMs in industrial settings. Additionally, most papers on confidence assessment use only illustrative examples to demonstrate their method, which are not sufficient to make broad claims about a method's generalizability or efficacy.

[1] It is worth noting that other authors such as Denney et al. and Asaadi et al. have expressed ideas about the functional purpose of DACs, but to my knowledge, have not distilled them into a singular definition.

1.5.2 Limitations of Quantitative CAMs

Quantitative CAMs are compelling in so far as they reduce complex assurance rationale(s) into one (or a few) numerical valuations that can be used as the basis for decision-making, either by interest holders or as part of (semi-)automated processes towards continuous or perpetual assurance. However, in addition to a lack of empirical data on their adoption, there are several theoretical limitations associated with quantitative CAMs.

Limitation: Interpreting Uncertainty Measures

Quantitative CAMs typically represent confidence in an AC's top-level claim as one or more numbers in the range $[0, 1]$. The results can easily be mistaken for probabilities of failure by those who are used to dealing with such probabilities, as has classically been the case in critical systems engineering[2]. In fact, these numbers are subjective probabilities representing degrees of belief or uncertainty: they should not be interpreted as probabilities of system success or failure from a frequentist perspective. For instance, for CAMs using Bayesian Networks as a formalism [48], an overall confidence of 0.96 in an AC's top-level claim does not mean that 4/100 times, the system will fail to satisfy the top-level claim; rather it should be understood as a subjective measure reflecting an expert's degree of belief that an AC's claim is true. Yet, when non-expert interest holders encounter numbers that look like probabilities their tendency is to interpret them as such. In short, "numbers carry" and lose the associated nuance and context over space and time[3].

Limitation: Over-Precision Fallacy

Even if issues of interpretation are overcome, quantitative confidence measures remain vulnerable to the fallacy of over-precision [80]. That is, an individual interpreting the confidence measure might find the measure more credible simply because it can be expressed to many decimal places (e.g., 0.826374 v. 0.8). In some fields such precision is well-founded if the underlying measurements used as input to a calculation are also extremely precise. However, in the case of AC confidence assessment, the inputs are

[2]Anecdotally, I have seen several otherwise very accomplished practitioners jump to this interpretation in industry. In fact, it was also my initial reaction when I first encountered these methods.

[3]This sentiment was raised by experts at an annual meeting of the Software Certification Consortium (SCC) in Annapolis, MD, USA in May 2024.

(almost) always expert judgement over complex engineering artifacts or assurance arguments, which are unlikely to be measurable with high precision. Experienced practitioners might be aware of this fallacy and take steps to prevent it; however, existing CAMs do not have "built-in" guards against it.

Recent work by Idmessaoud et al. has addressed this concern by defining a procedure that translates the numerical values produced by their method onto a qualitative scale for interpretation by end-users [50]. However, other methods still only produce numerical values that the user must directly interpret.

Limitation: Comparing Valuations Produced by CAMs

A challenge arises when comparing valuations produced by quantitative CAMs. When numerical scales are used, it is necessary to understand if the difference between two values is significant enough to merit an action. For instance, suppose initially a CAM produces a subjective probability valuation of 0.87 and then, after further improvements are made to the AC, the valuation rises to 0.89. While the confidence has increased, it is not clear if this difference is meaningful. Frequentist statistics has established procedures for comparing values based on sample sizes, but analogous procedures have not been developed for the currently available quantitative CAMs.

Limitation: Confidence Thresholds and Decision-Making

As noted above, quantitative CAMs can have a role in realizing DACs, especially when a DAC is used as an input to (semi-)automated decision processes. A quantitative CAM will produce *a* valuation of confidence, but that alone is not enough to decide whether to deploy or continue to operate a system. When a human is responsible for the decision they must interpret the valuation. If the decision is automated, then criteria must be established to make the decision. In either case, it is necessary to apply a threshold to the confidence valuations (or some secondary measure derived from the valuations), but how does one decide on the threshold?

Limitation: Flexibility in Reasoning

Quantitative CAMs use various mathematical operations to propagate confidence valuations from the evidence at the leaves of the argument to the top-level claim. Existing methods each define two or three operations. To configure belief propagation, the user selects one of these operations for each argument step in the argument. For

instance, the Bayesian Belief Network method described by Hobbs and Lloyd defines two such operations: "NoisyAND" and "NoisyOR" that the user must select from [48]. However, reasoning in an assurance argument can be more complex and is not always easily represented by (noisy) logic gates. For instance, it would be difficult to express the reasoning: *Once confidence in claim X is high enough, then all that matters is confidence in claim Y.*

Limitation: Method Validity and Trustworthiness

CAMs are ultimately intended to support decision-making about critical systems, where there are significant consequences if an accident occurs. Graydon and Holloway point out that none of the published CAMs have had their "trustworthiness" empirically validated [80]. Some methods have been subjected to sensitivity analyses to show that (on small examples) the confidence in the top-level claim of the AC changes as the inputs change [50]. While it is necessary for a method to be sensitive to changes in the input, this is not sufficient to show efficacy. Additionally, Graydon and Holloway found that many of the existing methods are vulnerable to counter-examples that produce unexpected results [80].

1.5.3 Missing Support for Dialectic Reasoning

As mentioned above, dialectic reasoning is one strategy for mitigating bias in ACs, particularly confirmation bias. Therefore, there is an interest in incorporating dialectic elements (e.g., defeaters) into ACs and confidence assessment. Indeed, some early attempts have been made to take account of dialectic reasoning using Baconian probabilities [81]; however, the efficacy of these calculations is uncertain [53, 82]. Defeaters may be raised when using the Assurance 2.0 method, and unresolved defeaters are to be considered in the method's "residual perspective". However, Assurance 2.0 does not appear to incorporate defeaters into the overall confidence calculation [46]. Quantitative methods, such as those based on Bayesian Belief Networks (BBNs) or Dempster-Shafer Theory (DST), do not take account of dialectic reasoning when computing confidence measures.

To stay abreast with the latest developments in AC argumentation, CAMs must account for dialectic reasoning. In real-world ACs, there are often reasons to doubt the argument, and (for various reasons: budget, schedule, limitations of knowledge, system novelty etc.) some sources of doubt go unresolved [53]. Therefore, to be

useful to practitioners, a CAM must account for reductions in confidence arising from unresolved defeaters.

1.5.4 Confidence for Dynamic Assurance Cases

As discussed in Section 1.4.3 above, DACs might have their assurance rationale change, their evidence updated, and be instrumented with indicators to validate their claims. Proposals exist for managing these aspects in isolation. For instance, methods exist for interpreting the impact of a specific indicator on a claim or local region of the AC [76, 63]. However, no method exists to compute an overall confidence assessment for a DAC based on these types of changes.

1.6 Research Goal

The goal of this dissertation is to develop a new CAM that allows practitioners to implement DACs for critical systems. The new method should also address the limitations formulated above. Towards this goal, the resulting CAM should satisfy the following criteria:

1. **Barriers to Adoption** - Address barriers that deter AC practitioners from assessing confidence in an AC.

2. **Quantitative Limitations** - Address known problems of quantitative CAMs, including: limitations related to interpretation, precision, comparison among results, flexibility in reasoning, and decision thresholds.

3. **Dialectic Reasoning** - Support dialectic reasoning in the AC.

4. **Dynamic Assurance** - Support dynamic assurance by enabling reasoning about the impact on confidence from changing arguments (i.e., assurance rationale), updated evidence, and changes to indicators.

Additionally, this dissertation will aim to show that the proposed CAM is trustworthy in the sense that it can be depended upon by practitioners to support decision-making about critical systems. Graydon and Holloway emphasize the importance of trustworthiness for CAMs [80]. However, they do not provide an actionable definition of trustworthiness and instead conclude that the "burden of proving that an assurance

argument confidence quantification technique produces trustworthy assessments lies with its proposers" [80]. Criteria for trustworthiness must look beyond "correctness" because it is not possible to show that a CAM produces "correct" results. There might not be a single correct answer since confidence is a matter of judgement, and different interest holders evaluate confidence differently based on their knowledge and experience. This means it is difficult to produce a single ground truth to use as a basis for evaluation. As an alternative, this dissertation will argue that the proposed CAM satisfies five elements that contribute to trustworthiness:

- **Applicability** - The proposed CAM can be used on a DAC for a representative critical system.
- **Understandability** - Users of the CAM can understand how confidence is propagated through the AC's argument and can interpret the results.
- **Sensitivity** - When changes are made in the confidence valuations at leaves of an AC's argument, the overall confidence assessment changes predictably.
- **Propagation Stability** - When confidence valuations are propagated through the AC's argument, they are not unnecessarily attenuated or amplified by the propagation procedure.
- **Intuitiveness** - The method produces results that align with a user's intuition and judgement about the overall confidence in the AC.

1.7 Approach and Contributions

The overall contribution of this dissertation is *Certus*, a method and domain-specific language for assessing confidence in dynamic assurance cases. *Certus* permits users to express confidence using vague, but linguistically meaningful, statements (e.g., "I have high confidence in X"). These are then propagated through an argument according to propagation operations specified using the language. A DAC annotated with expressions in the *Certus* language may be used as part of a continuous or perpetual assurance process, as shown in Figure 1.1, where the DAC is used by a (potentially automated) decision maker as part of a larger self-adaptive system.

Given a DAC expressed using a structured notation, *Certus* extends the model with three elements, shown as annotations on top of a small argument fragment in

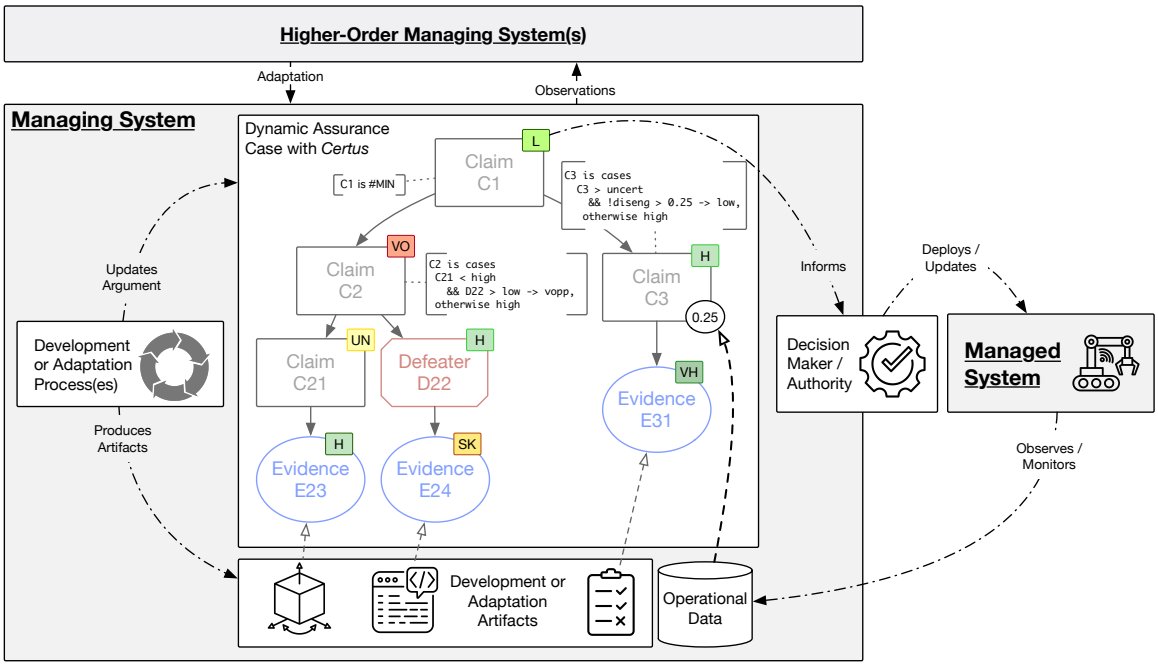


Figure 1.1: A DAC annotated with *Certus* expressions in a self-adaptive system.

Figure 1.1. First, nodes in the argument are annotated with expressions written in the *Certus* language that describe how confidence is propagated from a node's children to its parent. For example, see the expressions annotating Claims `C1`, `C2`, and `C3` in Figure 1.1. Second, the confidence valuations (e.g., *very opposed*) themselves appear on the nodes, indicating the computed level of confidence (per the *Certus* expression) or the user's assigned level of confidence (for leaf nodes). These are shown as coloured annotations on each node. Finally, indicators and artifacts are linked into the DAC. The DAC is developed and adapted as part of a larger dynamic assurance process carried out by a managing system. Higher-order managing systems might also exist to detect and respond to situations where the main managing system is inadequate.

1.7.1 Research Methodology

The research presented in this dissertation was developed using a multi-step methodology aimed at creating a novel CAM that is both trustworthy and addresses the needs of AC practitioners. This work also builds on my own professional experiences preparing and assessing ACs in an industrial setting across multiple industries: several of the CAMs published in the literature do not appear to be used in practice. The work was undertaken in three steps as discussed below.

Step 1: Requirements Elicitation

To begin, this research used a grounded-theory [83] approach to understand the current state of practice for confidence assessment among AC practitioners, including identifying barriers or limitations of existing methods. This was complemented by observations about theoretical limitations, as described above in Section 1.5.2. Using this information, requirements were elicited from three perspectives: 1) the Practitioner Perspective, 2) the Dynamic Perspective, and 3) the Dialectic Perspective. These requirements were used to drive the development of a new mixed (qualitative and quantitative) CAM called *Certus*.

Step 2: Developing *Certus*

The development of *Certus* was approached in two steps: 1) creating the *language* for specifying confidence propagation and valuations, and 2) creating a *method* to guide the use of the language.

Developing the Language. A denotational semantics for the *Certus* language was specified using the meta-programming capabilities of the Lean4 theorem prover. The formal semantics provides an unambiguous interpretation of expressions in the *Certus* language and contributes to the trustworthiness of the resulting confidence assessments. To illustrate the language's syntax and semantics, and as a means of preliminary validation, it was applied to a small example from the automotive industry. To support more substantial evaluation, a parser for the language was implemented in TypeScript as part of an open-source library that was subsequently integrated with *Socrates*, a commercial AC management tool[4]. Both implementation activities provided further opportunities to refine the language and provided preliminary validation of the concept.

Developing the Method. The *Certus* method describes how to apply the language to assess confidence in a DAC. The method was developed to align with the larger AC method, spanning the development, evaluation, and deployment (and operation) of a critical system. To both demonstrate the method, and as a means of preliminary validation, it was applied to a small example from the automotive industry.

[4] `https://criticalsystemslabs.com/socrates-assurance`

Step 3: Evaluating *Certus*

Certus was evaluated to determine if it satisfies the requirements for a new CAM elicited from Step 1, and to gather evidence to show that the criteria from the research goal are satisfied. Four dedicated analyses were used to evaluate properties of the language: 1) a propagation analysis was used to check propagation stability; 2) a scalability analysis was used to estimate the level of effort required to apply *Certus*; 3) a sensitivity analysis was used to investigate the sensitivity of *Certus*' propagation operations; and 4) an expressivity analysis was used to show that the language can be used to express a range of confidence propagation operations. Finally, a case study applied both the *Certus* method and language to assess confidence in a DAC for a (prototype) medical device.

1.7.2 Contributions

This dissertation makes several conceptual and practical contributions. Some of these contributions have already been partially published: [57, 84, 52, 85]. Other results appear for the first time here, including: the formalization of the language in Chapter 5, the *Certus* method in Chapter 6, and the results of the evaluation in Chapter 7.

Conceptual Contributions

Beyond the contribution of the *Certus* method and language, this dissertation makes four broader contributions to knowledge in the field of confidence assessment for ACs.

The Practitioner Perspective. The current state of practice of AC confidence assessment is synthesized from interviews with 19 AC practitioners, including identifying CAMs that are currently in use and barriers to adopting new methods [52]. This contribution will help researchers working in the field of AC confidence assessment develop, or adapt, methods to meet the needs of practitioners. For practitioners, this perspective provides a view into the methods used, and challenges faced, by their peers across different industries.

The Dialectic Perspective. It is shown that dialectic reasoning (i.e., "defeaters") can be included in the computation(s) performed for quantitative CAMs. Three different interpretations of dialectic reasoning in ACs are identified: the deductive, eliminative, and strict interpretations. Eleven requirements for including defeaters in these assessments are defined. The requirements are used to extend an existing

quantitative method to account for defeaters [84], and then used to build dialectic reasoning into *Certus*. For researchers, this contribution provides requirements for incorporating dialectic reasoning into quantitative CAMs they are developing or improving. For practitioners, this contribution outlines different interpretations they may apply when considering defeaters in an AC, either quantitatively or qualitatively.

The Dynamic Perspective. It is shown that a CAM (namely *Certus*) can be used to assess confidence in a DAC containing indicators connected to "live" data sources and changing evidence. Confidence in a DAC can be (re-)assessed as these elements change, and the assessment results can be used to support deployment and operating decisions for a system. For both researchers and practitioners, this contribution offers a means to integrate confidence assessments with dynamic assurance processes which could be incorporated into work on continuous or perpetual assurance.

The Linguistic Perspective. It is shown that confidence in a (D)AC can be represented using vague, but linguistically meaningful, expressions that may be interpreted both qualitatively and quantitatively. Moreover, it is shown that the logic to propagate confidence through an argument can be expressed using a domain specific language to reflect the nuanced reasoning of argument structures in the confidence propagation logic. For researchers, this contribution offers a new perspective on quantitative confidence assessment, compared to existing quantitative CAMs that use mathematical formulae to describe propagation.

Technical and Practical Contributions

In addition to the conceptual contributions discussed above, this dissertation makes several technical and practical contributions:

1. A synthesis of existing methods and notations used to prepare and maintain ACs into an overall methodology for developing ACs. This overall method is used as the basis for the *Certus* method.

2. Extending the Bayesian Belief Network (BBN) CAM to account for dialectic reasoning [84]. Extending the BBN method provided an opportunity to validate the requirements of the dialectic perspective.

3. A formal specification for the *Certus* language authored in the Lean4 theorem prover. The specification provides an unambiguous interpretation of every statement in the language, supporting the trustworthiness of the language.

4. A case study demonstrating the application of *Certus* to an (D)AC for a prototype medical device. This case study was one of five evaluation activities.

5. An open-source TypeScript library called `certus-ts` that implements the *Certus* language, including a parser and interpreter[5]. The implementation provided preliminary validation of the language concept and was used for additional evaluation activities.

6. A prototype implementation of *Certus* in *Socrates*[6], which provided validation of the language concept as part of the case study.

1.8 Reading Guide

This dissertation might be of interest to four different audiences: researchers working on CAMs, researchers working on ACs, systems and software engineering researchers, and AC practitioners. Figure 1.2 suggests a reading sequence for each audience.

CAM Researchers. For researchers working on AC confidence assessment, particularly on quantitative methods, most chapters are recommended. To the extent that CAM researchers are already familiar with the AC method, Chapter 2 may be read lightly. The formalization of the *Certus* language Section 5.4 could be read lightly by those who are not interested in the detailed semantics of the *Certus* language.

AC Researchers. For researchers working with (D)ACs, but who are not necessarily focused on confidence assessment, Chapter 2 might be known material and may be read lightly. Chapter 3 is recommended as it describes the practitioner perspective on CAMs, which is related to other areas of AC practice. Some sections in Chapter 5 may be read to develop a sense of the *Certus* language, particularly Sections 5.1 (orientation to the language), 5.6 (conditioning on indicators), and 5.7 (conditioning on artifacts). The model for representing belief in an AC using fuzzy sets in Section 5.3 might also be of interest. The *Certus* method in Chapter 6 is recommended reading, especially as the method relates to the larger AC method. The demonstration of the method provided in the case study in Section 7.5 might also be helpful in this regard.

[5] `https://www.npmjs.com/package/certus-ts`

[6] I am currently employed by Critical Systems Labs Inc., the company that maintains and sells the *Socrates* tool. To avoid a potential conflict of interest, the specification and implementation of the *Certus* language in the `certus-ts` library has been made available under an open-source license. The *Socrates* tool is a solution for managing ACs that can be used without *Certus*.

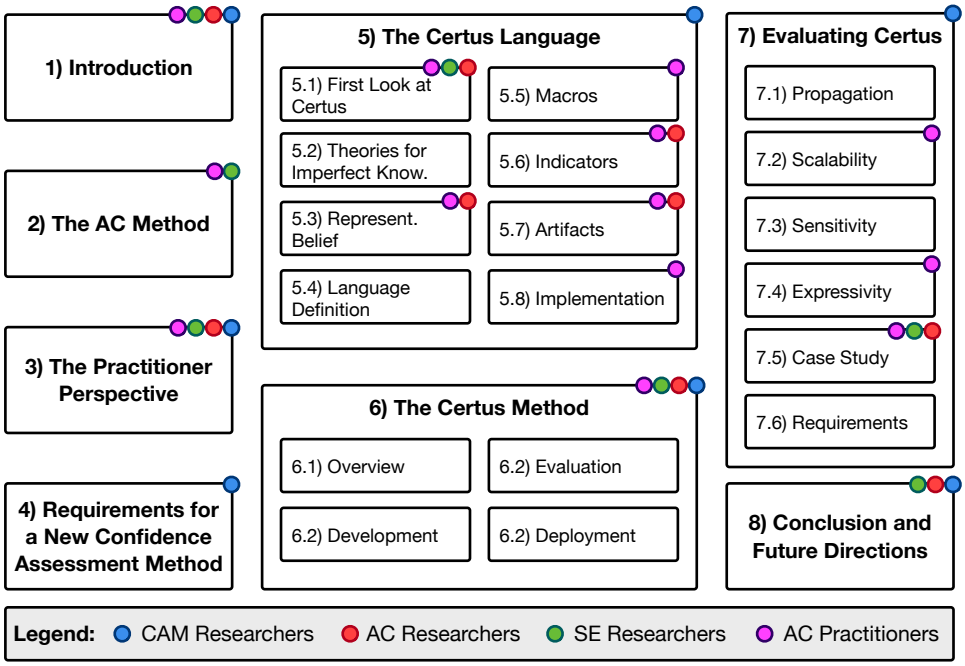


Figure 1.2: Reading guide for different audiences.

Systems and Software Engineering Researchers. For researchers working in the broader systems and software engineering communities, Chapter 2 describes the foundational elements of the AC method. The practitioner perspective in Chapter 3 might be of interest, especially in so far as the challenges faced by AC practitioners are related to challenges from other fields of systems and software engineering. Section 5.1 provides a brief orientation to the *Certus* language. The method description in Chapter 6 and the case study in Section 7.5 connect to aspects of larger system or software engineering processes.

AC Practitioners. For AC practitioners, especially those considering CAMs, Chapter 2 provides an overview of the AC method, and might offer pointers for future reading. Chapter 3 could be read to understand how others in the field approach confidence assessment. In Chapter 5, the overview of the language in Section 5.1 is a good starting point, perhaps followed by the belief model described in Section 5.3. Sections 5.6 and 5.7 are relevant to DACs. The method in Chapter 6 describes how to use *Certus* with a (D)AC. The scalability analysis in Section 7.2 might be of interest when considering the level of effort required to apply different CAMs and the expressivity analysis in Section 7.4 identifies common patterns of use for the language. The case study in Section 7.5 provides a worked example for reference.

Chapter 2
The Assurance Case Method

As introduced in Chapter 1, assurance cases are used to reason about essential quality attributes (e.g., safety or security) of critical systems across a wide range of industries. However, no single method is widely recognized as *the one* "assurance case method", covering all aspects of AC development, evaluation, and monitoring. In practice, organizations compose available notations, methods, and techniques to develop their own "bespoke" AC methodology that is tailored to their specific needs[1]. Nonetheless, common themes have emerged across different uses of ACs, and proposals for end-to-end methodologies exist [21, 86, 46, 45, 87, 61, 24]. Collectively, they describe core aspects of an "assurance case method" that this dissertation builds upon. This chapter introduces foundational concepts by describing that method and reviewing related work on confidence assessment and dynamic assurance.

2.1 Overview of the Assurance Case Method

Figure 2.1 outlines the main elements of the method for developing and evolving ACs. At its center is the assurance case, which is viewed as a live model of assurance. All activities undertaken as part of this method operate on, or use, this central model in some manner. There are three *phases*: Development, Evaluation, and Deployment, each containing *steps*. Though the method is depicted as a progression through the steps and phases, in practice it is common to move back and forth between phases[2]. For instance, after assessing the argument, it is possible that new arguments need to be marshalled to address the identified challenges, necessitating a return to the Development phase. Moreover, some sub-methodologies are applicable in multiple

[1] This is an observation from my own experience. However, even light reading of relevant literature reveals a number of different conceptualizations of the "assurance case method".

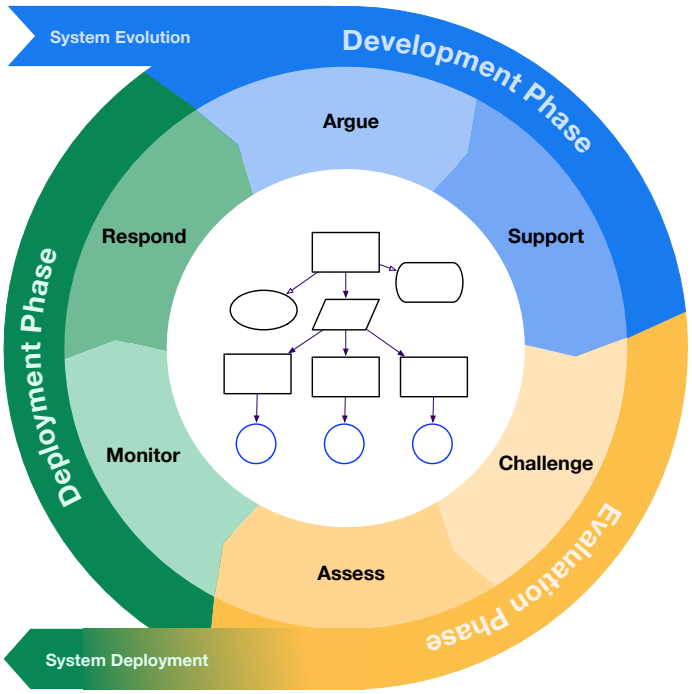


Figure 2.1: Overview of the assurance case method.

steps, as will be discussed below. Nonetheless, this depiction of the AC method serves as a framework for organizing and introducing foundational concepts in the AC method.

An AC begins its life in the **Development** phase. Sometimes ACs are prepared as part of a "wrapping up" activity, near the end of a system's (initial) development. However, ideally AC development begins early in the engineering lifecycle, possibly as part of early concept formulation, and continues in parallel with system development activities [88, 58, 45]. This practice is sometimes referred to as "assurance-based development" (ABD) [88]. Regardless, development begins with the **Argue** step, wherein the rationale for assurance is described in a (semi-)structured form. Then, in the **Support** step, the argument is connected with supporting evidence. Depending on the methodologies employed, this step might also involve preparing extra "confidence arguments" that support the main argument.

Once an argument has been created, the **Evaluation** phase begins with the **Challenge** step. In this step, counter-arguments, or reasons to doubt the main argument (i.e., "defeaters"), are presented in a dialectic[3] manner. If rebutting arguments are necessary, or the overall argument must be revised due to the discovery of a "fatal

[2]The resulting diagram would be rather difficult to read if every transition was accounted for!

defeater", then it is necessary to return to the Development phase. Subsequently, the **Assess** step is undertaken, which usually involves a combination of (independent) review and systematic methods for assessing the argument and evidence.

If the AC is assessed to be acceptable, then a decision is made to deploy or otherwise operate the system, and the **Deployment** phase is entered, beginning with the **Monitor** step. During monitoring, data from the system's operation is collected to validate claims made by the AC's argument (see Section 2.2.3). If monitoring reveals that a claim in the AC is invalid, then the **Respond** step is triggered. The AC, and possibly the system, must be modified, which returns to the Development phase.

As discussed in Chapter 1, ACs are increasingly viewed as live models of assurance. Therefore, when the underlying system is changed, the AC must also be changed. Change in the underlying system might be part of system development activities performed by humans, or undertaken (semi-)autonomously as part of a self-adaptive system's adaptation procedure. Changes to the AC due to system evolution trigger a return to the Development phase to revise the argument and evidence.

Relation to Other Process Models and Architectures. The AC methodology outlined above has many similarities with existing process models for system and software development and operations. For example, its iterative nature is similar to an Agile software development process, and the Deployment phase contains elements of the "DevSafeOps" process model [61]. There is also a synergy with process and architectural models used by self-adaptive systems. For instance, this method could be embedded within a Monitor, Analyze, Plan, and Execute, with shared Knowledge (MAPE-K) feedback control architecture as a means of maintaining a dynamic assurance case for a self-adaptive system [75, 67].

2.2 The Development Phase

Developing a new AC begins with crafting a top-level assurance claim about the system. Often the claim is of the form: *The [System] is acceptably safe* (or secure, etc.), though it is prudent to articulate a more precise claim that bounds the scope of the AC more tightly. The phrase "acceptably safe" can be interpreted in several ways

[3]The Oxford English Dictionary defines *dialectic* as a "critical investigation of truth through reasoned argument, often spec. by means of dialogue or discussion." [89].

[21], though a common interpretation is that the risk associated with the system is *As Low As Reasonable Practicable* (ALARP).

The Development phase is focused on preparing an evidence-supported argument that demonstrates the truth of the top-level claim. During the Development phase, the **Argue** and **Support** steps of the AC method are often intertwined. This section discusses foundational concepts, methods, and notations related to creating and supporting AC arguments.

2.2.1 Notation for Expressing Assurance Arguments

There are several ways to express an assurance argument. Historically, the assurance rationale was captured in a narrative style "Assurance Report", sometimes called a "Safety Report" or "Security Report". This approach is still common in some industries, such as nuclear power generation. While the narrative approach offers significant flexibility in terms of how the case is expressed, it does not enforce a systematic approach to reasoning about assurance, and does not scale well as systems become more complex [27]. Moreover, a document-based approach is onerous to maintain as the underlying system changes, which is increasingly the case for critical systems.

As an alternative to a narrative style report, Kelly [27] and Bishop and Bloomfield [28] proposed structured argument notations that visualize arguments as "box and arrow" diagrams. Their notations adapted a generic notation for expressing philosophical arguments developed by Toulmin [30]. Since their introduction in the late 1990s, structured notations have become widely used to express and organize assurance arguments, and are the foundation of many developments in AC methodology in the last twenty years, including for confidence assessment and dynamic assurance. In addition to increased rigour, they also provide a basis for automation and tool support, enabling model-based engineering practices for AC management [90].

The remainder of this sub-section introduces the Eliminative Argumentation (EA) notation, which is used for the rest of this dissertation. EA was selected for this work for three reasons. First, the names of the nodes (claim, evidence, etc.) naturally align with every-day terminology for describing arguments. Second, it was the first notation to introduce dialectic reasoning, which is a central theme in this work. Finally, I am familiar with it, having used it for a number of industrial projects over the last eight years (e.g., [53, 58, 91]).

The Eliminative Argumentation (EA) Notation

Like many structured notations, EA positions the top-level claim of the argument at the top of a diagram, as shown for `C1` in Figure 2.2. The claim contains an assertion that "It will snow tomorrow in Victoria"[4]. Some additional context for this claim is provided in `X2`.

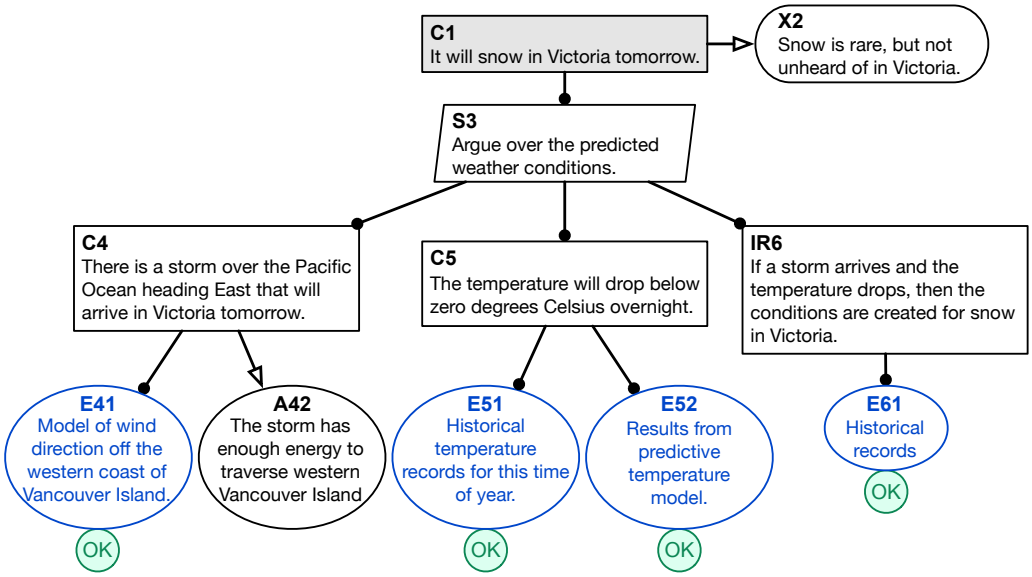


Figure 2.2: An example of a small argument about the weather in Victoria, B.C. expressed using the Eliminative Argumentation (EA) notation.

Claim `C1` is supported by strategy `S3` that describes how the parent claim is decomposed into sub-claims[5]. Strategy nodes are not strictly required, but are encouraged for situations where the decomposition approach might not be obvious to the reader. In this case, the strategy is to argue over the predicted weather conditions that usually lead to snow in Victoria, B.C. The context node `X2` provides additional information that can be helpful for interpreting the claim, in this case some context related to the normal weather in Victoria is provided. Sub-claims `C4` and `C5` assert the specific weather conditions that are predicted to occur.

The decomposition of a parent claim into supporting child claims (with or without a strategy), is referred to as an *argument step*. Other types of nodes might be involved in argument steps as well, such as evidence nodes or defeaters. An AC's argument

[4]As I am writing this, the weather forecast is calling for "Wintry Mix" for Victoria, B.C., which apparently means some combination of snow and rain.

[5]Goodenough et al.'s original EA notation did not contain strategy nodes. However, given their use in similar notations, strategies have been adopted by EA as well [57].

may be thought of as the composition of many argument steps where each step can be reasoned about in isolation, assuming the children are shown to be true by their respective argument steps. When referring to an EA argument step inline in text, the following notation is used: `C1` —• `C4`,`C5`, with the parent's identifier on the left-side and the children listed on the right side.

The inference rule (`IR`) node type is complementary to the strategy node type in EA. Where a strategy provides a generalized "top-down" perspective of how to decompose an argument step, the inference rule provides the "bottom-up" perspective describing how the children combine to show that their parent claim is true. In the argument in Figure 2.2, `IR6` provides additional rationale for why the specific combination of cooler temperatures and inbound precipitation is likely to result in snow. Inference rules are always present, though sometimes they are implicit because the inference from children to parent is thought to be obvious. A benefit of using an inference rule is that it can be both supported, by its own sub-argument, or challenged by an undercutting defeater, as will be discussed in Section 2.3.1 below.

Each of nodes `C4`, `C5`, and `IR6` is the root of its own argument step, supported by evidence. For instance, `C4` is supported by computer modelling results for the wind direction near Victoria (`E41`). An `OK` annotation is provided below `E41` to indicate the line of reasoning is complete; other terminators will be discussed below. An unsupported assumption (`A42`) also appears under `C4` that the storm will not lose energy as it passes over western Vancouver Island on its way to Victoria.

EA Node and Edge Types. As shown above, EA has several node types and two edge types. These can be organized into a type hierarchy shown in Figure 2.3. Structural node types are used to compose the main lines of reasoning in an argument. Among the structural nodes are several sub-types, most of whose roles were discussed above. Claims and Evidence nodes are both Premise-typed[6] nodes. Positive support for a Claim may be provided by sub-Claims or by Evidence. Strategy nodes support Claims by providing a decomposition. In turn, Strategy nodes may be supported by sub-Claims in accordance with their decomposition.

Inference Rules support Premises by rationalizing how a parent Premise is supported by its children. In turn, Inference Rules can be supported by Premises to justify their rationale, like how `E61` supports `IR6` in Figure 2.2.

Completed lines of reasoning in the argument are terminated with an `OK` node. If

[6]Adopting a term from Holloway's FAN notation, where nodes that support a conclusion are called a "premise" [32].

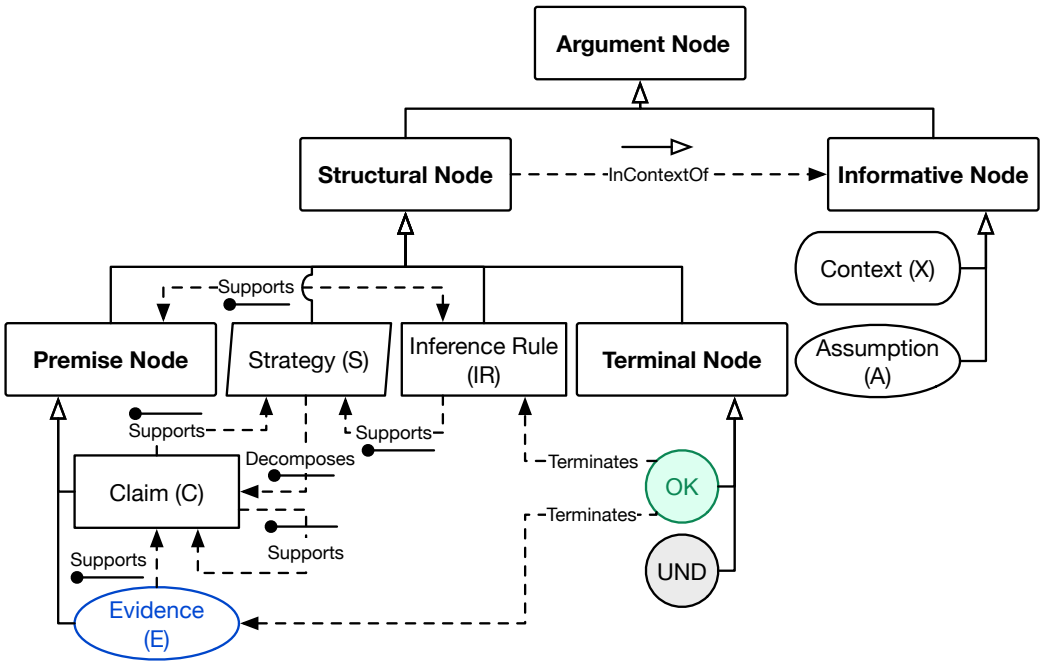


Figure 2.3: Type graph for node and edge types in EA. Generalizations (as in UML) are shown with unshaded arrow heads. EA edge types are shown with dashed lines and solid arrow heads, annotated with the type of relationship and the symbol used to denote the edge type in EA.

an Inference Rule is accepted "as is", it is terminated by an `OK` terminator. If there is remaining work to be done to complete a line of reasoning, then the `UND` terminator is used to indicate the node is "undeveloped" (not shown in Figure 2.3). All lines of reasoning must be terminated, otherwise the argument structure is invalid.

Informative nodes are not essential parts of an argument's main line(s) of reasoning, but provide additional contextual information that aids interpretation. They connect to structural nodes via the `InContextOf` edge type.

Defeaters are a central part of EA, and are conspicuously absent from the type graph in Figure 2.3. These will be introduced in full in Section 2.3.1, accompanied by a type graph showing their relationship to other EA node types.

Formalization of the EA Notation as a Graph. EA represents an assurance argument as a series of typed nodes connected by typed edges to form a directed acyclic graph (DAG), which can be modelled as a tuple $G = (N, E, \tau^N, \tau^E, \texttt{id}, \texttt{text})$ with:

- N is the set of nodes in the graph.

- $E \in (N \times N)$ is a relation describing the edges in the graph. An edge $(u, v) \in E$ means that there is a directed edge between node u and node v. The relation E is often modelled as an adjacency list, as will be done in Chapter 5.
- $\tau^N : N \to \texttt{NodeType}$ is a function mapping the nodes to their corresponding node type: $\texttt{NodeType} = \{\texttt{Claim}, \texttt{Evidence}, ...\}$.
- $\tau^E : E \to \texttt{EdgeType}$ is a function mapping the edges to their corresponding edge type: $\texttt{EdgeType} = \{\texttt{Supports}, \texttt{InContextOf}, ...\}$. For brevity, the edge type relation is omitted from many of the formalizations later in this dissertation because `Supports` is the main edge type of interest for confidence assessment.
- $\texttt{id} : N \to \texttt{String}$ is a function mapping each node to a unique identifier, which is often (but not always) a number. This is typically combined with its node type to refer to the node (e.g., `C1234` for a claim node with identifier `1234`).
- $\texttt{text} : N \to \texttt{String}$ is a function mapping from each node to a character string describing the text of the node's assertion (e.g., "It will snow in Victoria").

The above formalization for an EA graph can be extended to include other aspects that are relevant to assurance, such as associations to artifacts or indicators. This will be done in Chapter 5.

Mathematical Notation for AC Nodes. Throughout this dissertation, two different notations are used for referring to nodes, depending on the context. In situations where mathematical typesetting is used, nodes appear as c_{1234}, where the letter corresponds to the node type (e.g., "c" for claim) and the number is the node's unique identifier. In situations involving a concrete example, like the narrative supporting Figure 2.2 above, typewriter font is used (e.g., `C1234`).

Other Structured Notations for Expressing Assurance Arguments

Other structured notations for expressing ACs include: the Goal Structuring Notation (GSN) [27, 20], Claims-Argument-Evidence (CAE) notation [28], the Structured Assurance Case Meta-Model (SACM) [26], and the Friendly Argument Notation (FAN) [32]. Toulmin's notation is also occasionally used [30], but mainly as an illustration of first principles for argumentation.

Of these notations, GSN, CAE, and SACM depict arguments as diagrams with nodes containing assertions and arrows connecting between parents and children. The

GSN notation contains elements for modularizing arguments, which can be useful for managing large assurance cases in complex system integration projects involving multiple organizations. SACM also provides specifications for modularity, as well as constructors for modelling other related concepts, such as linking to supporting artifacts. The CAE notation defines node types that may be composed using a handful of generalized argument steps (e.g., "Concretion" or "Substitution") [46].

FAN takes a different approach, expressing arguments using structured text. The motivation is to use a minimal notation that focuses on the essential elements of an argument: premises and conclusions [32].

Many common themes exist among the available structured notations. For instance, a core idea represented in all notations is that arguments can be decomposed into "steps" and that each step can be analyzed. Though EA is used from here on to express ideas, the contributions of this dissertation do not rely on any specific features of the EA notation. In particular, the theoretical contributions will rely on two concrete node types from EA: Claims (more generally, Premises) and Defeaters, both of which have analogs in every other notation. Therefore, the results from this work will generalize to other notations.

2.2.2 Supporting Arguments with Evidence

The following quote[7] from ISO 262626, a widely used automotive functional safety standard, summarizes the interdependence between an argument and its evidence:

> "*An argument without supporting evidence is unfounded, and therefore unconvincing. Evidence without an argument is unexplained, resulting in a lack of clarity as to how the safety objectives have been satisfied.*" [13, Part 10, p. 12]

Clearly evidence is an essential element of an AC. In EA, and other notations, the evidence nodes are representations of the actual evidence. They act as a bridge, connecting the *artifacts* containing the actual evidence to the claims they are supporting [26]. A wide range of artifacts are produced in the course of developing a critical system, including: operational and system concepts documents, architectural descriptions, detailed design reports and specifications, source code (and similar

[7]The original version of this quote is from Tim Kelly's doctoral thesis [27, p. 25].

implementation artifacts), system models, verification records (reviews, etc.), test reports, simulation studies, formal proofs, analysis results, engineering process and procedure documents, and field reports.

Linking Artifacts to an AC

There is no widely used notation for showing relationships between artifacts and evidence nodes, though SACM does suggest a document icon, and similar notations are used by various tools. This dissertation shows associations between artifacts and evidence nodes in an argument similarly using an icon: , as demonstrated in Figure 2.4 with a fragment weather argument.

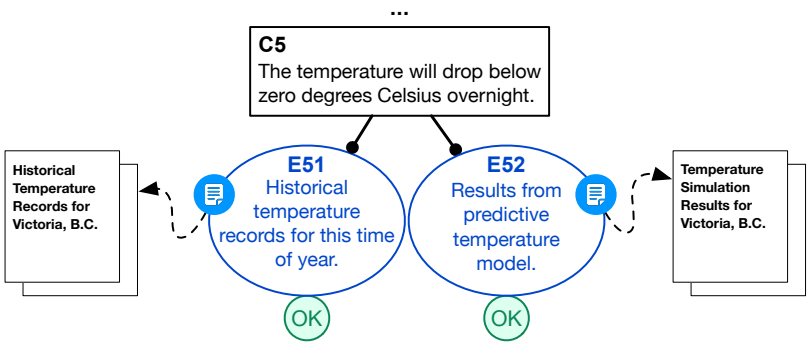


Figure 2.4: Example of an artifact connecting to evidence in an AC argument.

Evidence in Dynamic Assurance Cases

As mentioned in Chapter 1, the evidence (i.e., the artifacts) supporting an AC are subject to change in a continuous or perpetual assurance process. For example, in the ENTRUST method, new artifacts are generated as part of the adaptation process for a self-adaptive system [75]. Using artifacts to connect an argument to external models of system development (e.g., SysML models) is important for ensuring that an AC remains up to date with the latest evidence [90]. Artifacts are formally modelled in Chapter 5 to include them in a confidence assessment method.

2.2.3 Supporting Arguments with Indicators

Another way to support an AC is with streams of "live" data collected from the system's operation or development activities. These are referred to as *performance*

indicators, and sometimes the quality attribute of interest is added as a prefix (e.g., "safety performance indicator", often abbreviated to SPI).

A further distinction can be made between *leading indicators* and *lagging indicators* [92]. Changes to leading indicators precede loss events while lagging indicators only change after a loss event has occurred. For example, the number of "near-miss" events is often regarded as a leading safety performance indicator for a safety-critical system, and the number of accidents is a lagging safety performance indicator.

Indicators in the AC Method. Indicators could arguably be positioned in the Evaluation or Deployment phases of the AC Method outlined in Section 2.1. While it is true that they have an important role in both phases, in particular the Monitor step within the Deployment phase, the work of instrumenting an AC with indicators often occurs as part of AC development.

Anatomy of an Indicator

Denney and Pai suggest that an indicator has four fundamental parts [64]. **Measures** capture data about system operations or development activities. For example, the number of near-miss collisions with pedestrians, cyclists, and other road vehicles (each being its own measure). **Metrics** synthesize the measures to produce a representative value. For example, a "criticality-weighted near-miss" metric might weight near-miss events for pedestrians and cyclists more than other vehicles. **Thresholds** describe target values a metric should (or should not) reach over a period of **Exposure**. For example, the criticality-weighted near-miss metric should not exceed 10 units over a period of 30 days.

While keeping the spirit of Denney and Pai's definition, we generalize this construction slightly. For this dissertation, an indicator has three parts: 1) a **metric**, possibly derived from a synthesis of one or more measures; 2) a set of threshold **categories**, e.g., `Ok`, `Warning`, `Danger`; and 3) a **valuation function** that evaluates a metric's value(s) and outputs a category. For example, the valuation function might output `Ok` if our near-miss metric is equal to zero, `Warning` if the value is greater than zero in the last 30 days, and `Danger` if the value exceeds 9 units over the last 30 days.

Notably, our valuation function generalizes Denney and Pai's notion of a threshold and exposure, increasing flexibility and permitting multiple categories of "concern" to be used, rather than a single threshold. Additionally, a prerequisite for connecting an indicator to the AC is to formulate a *metric* from one or more *measure(s)*. Or-

ganizations make significant investments in the infrastructure for deriving indicators from vast amounts of data collected from the field.

Instrumenting an AC with Indicators

No widely accepted notation exists to connect indicators to elements of an AC's argument. This dissertation uses a similar iconography as for artifacts. When referring to an indicator in general terms, the icon is used, where traffic signal colours are used to represent the common categories (e.g., red means `Danger` and amber means `Warning`). If the value of the underlying metric is relevant, then it can be included as 3.4, as in Figure 2.5.

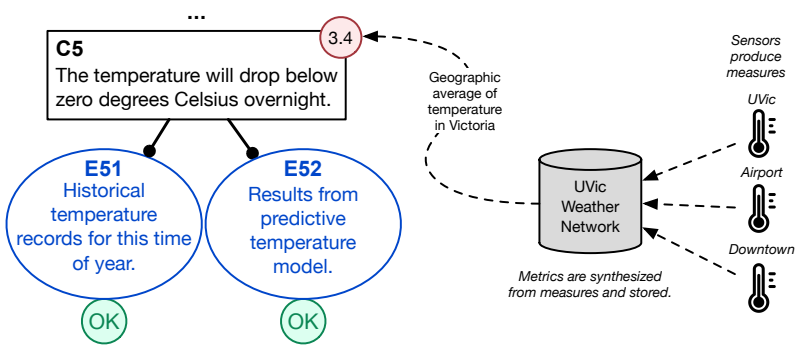


Figure 2.5: Example of an indicator linked to a node in an AC argument.

In the example, suppose that an indicator called `Freezing` is created to track the current temperature in Victoria, B.C. as an average of several measures taken from multiple locations[8]. The measures are aggregated to produce an average temperature, T, for the area. The valuation function returns `Ok` for $T \leq 0$, `Warning` for $T > 0 \wedge T \leq 1$, and `Danger` otherwise. In this example, since the current temperature is 3.4 degrees Celsius, the indicator is evaluated to be in the `Danger` category.

Dynamic Assurance Cases with Indicators

As identified in Chapter 1, instrumenting an AC is an important theme in the literature on dynamic assurance: instrumenting indicators help to elevate it to a DAC that provides "through-life" assurance for a system. When indicators are associated

[8]As a point of interest, the Victoria School-Based Weather Network is a long-term project run by researchers from UVic that aggregates data from many sensors located at public schools, among other locations [93]. See `https://victoriaweather.ca`.

to nodes in an AC, they provide a means to continuously validate the assertion within the node. As seen in the weather example above, an indicator can be used to show if a claim holds based on live data. Alternatively, if defeaters are used to challenge the case, then indicators can be used to determine if a hypothesized defeater is credible.

Symbiotically, the node and its argument context show why that indicator matters to the overall assurance argument. If the value of the indicator changes unfavourably, then one can quickly see how it impacts the overall assurance claim by tracing upward within the argument's structure. This offers an improvement over the situation that often occurs with "data dashboards" that display a plethora of metrics, but do not necessarily describe why those metrics, or corresponding thresholds, matter.

The examples given focus on using indicators to measure properties of the environment (e.g., temperature) or a system's behaviour during operation (e.g., near-miss events). However, indicators can also connect data from system development activities to an AC. This could be useful as part of a continuous assurance process where the decision to deploy the next release of the system depends on data generated by development teams. For instance, the complexity of a critical software component might be captured by an indicator synthesizing measures like cyclomatic complexity, module coupling and cohesion, the number of source code lines, and so on. If this measure suggests the software component's source code has become too complex, then assurance claims about the simplicity of the software's implementation or design might be invalidated.

2.2.4 Automated Generation of ACs

For real-world systems, ACs can become quite large, consisting of thousands of nodes. Creating these arguments can be a time-consuming process, and when multiple developers are involved variability can be introduced in terms of argument style. Moreover, organizations that manage multiple similar ACs (e.g., as part of a product line), often rely on similar assurance arguments, and so creating each AC "from scratch" is not necessarily a good use of resources. As a result, there is interest in methods that enable the automated generation of ACs. While methods for generating ACs are not foundational for this dissertation, it is an important area within the larger AC method and an active area of research in the field, and so some of the main methods and concepts are summarized below.

Potential Consequences of Automation in ACs Development. Much the value in developing an AC is found in the process of systematically reviewing, organizing, and presenting assurance-related information. To achieve the full value of an AC as a means for managing risk in a critical system, developers must deeply engage with the activity of preparing and supporting their arguments. In short, one must "do the work"[9]. There is a danger that automating AC generation will undermine the value that comes from experts critically engaging with a topic. This is not to say that we should avoid automation entirely: there are surely gains to be had, especially for repetitive or error-prone tasks. However, we must be thoughtful in its application, and ensure that we do not sidestep the *critical thinking* that is foundational to AC development.

Generation Using Patterns and Templates

In parallel with the development of the GSN, Kelly proposed that concrete arguments could be generalized into "patterns" that could be re-used [94, 27]. An extension to the core GSN was created to specify argument patterns containing *structural abstractions* [20, 95, 96]. When the pattern is instantiated into a concrete argument, the user resolves the structural abstractions with concrete values. There are three types of structural abstractions: *optionality* (a node or sub-argument is optionally instantiated), *multiplicity* (multiple instances of a node or sub-argument are instantiated), or *structural choice* (choose among several options to instantiate).

An example of a pattern using the EA notation containing all three types of structural abstraction is shown in Figure 2.6. The pattern is a generalization of the running weather prediction example. The structural choice (shaded diamond) abstraction requires the user to select two out of four of the possible weather conditions to argue over. There are also two multiplicity abstractions (shaded circle) requiring that the user select "1 or more" and "0 or more" instances of the node beneath. In this case the multiplicity abstraction is used to generate evidence nodes to reference multiple prediction models, one model per instantiated evidence node. Finally, the optionality abstraction (unshaded circle) allows the user to optionally include a context node.

One of the benefits of patterns is that they standardize argumentation. An organization or industrial community can develop a catalogue of patterns that encode

[9] I have been the author of several assurance arguments for critical systems and can attest to the value of "doing the work". At times it is tiresome, but every time my colleagues and I have done this, it has produced insights that we would not have found otherwise.

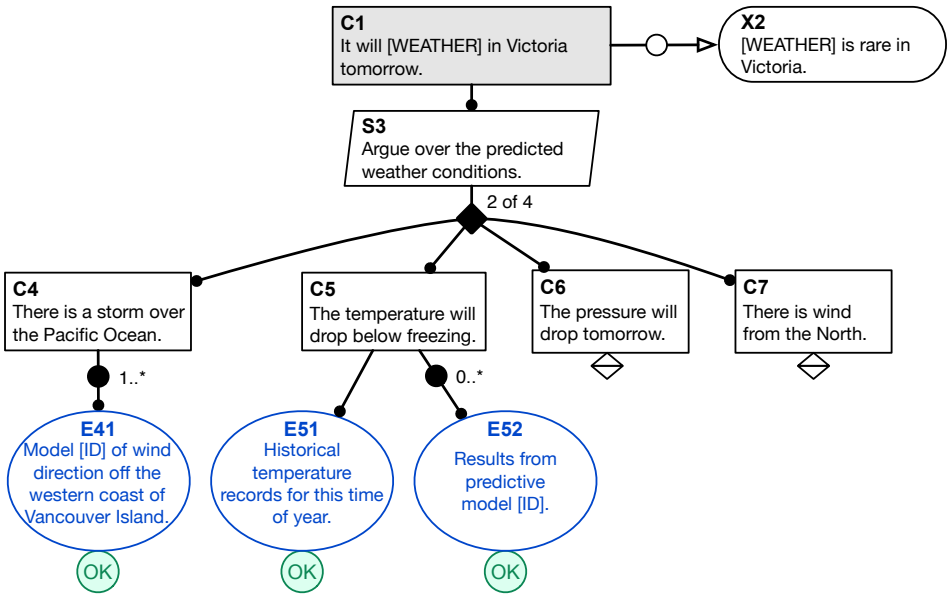


Figure 2.6: Example of an argument pattern in EA.

shared knowledge about how they undertake system assurance [27]. For instance, Chowdhury et al. proposed a principles-based approach to developing a catalog of patterns for compliance with ISO 26262 [97].

Generation from System Models

ACs are fundamentally about assuring a particular system (or organization, etc.). An AC is only one work product among many that are produced in the course of system development and evolution. For complex systems, maintaining consistency across the totality of work products is a significant challenge[10], with Model-Based Systems Engineering (MBSE) offering a solution. Instead of creating and maintaining many interdependent documents, including a document for the AC, digital models are developed and connected via tools. In the MBSE paradigm, structured AC arguments are part of the collection of models that describe a system. Several methods have been developed for generating, updating, or otherwise synchronizing, ACs with other system models that exist in an MBSE ecosystem.

Linking System Models to Enable AC Re-use. Kokaly et al. developed a method of connecting an AC to other system models (e.g., UML sequence and class diagrams) such that, when the system model changes, the affected elements of the

[10]Imagine the totality of work products that are produced for a modern road vehicle.

AC can be easily identified [98]. Their method enables (semi-)automatic extraction of elements of an AC that can be re-used when creating the AC for the modified system. They implemented their approach in a prototype tool called MMINT-A [90].

Workflow+ for AC Generation. Annable et al. developed the Workflow+ method that is capable of generating an AC from system models [99]. They created meta-models for several aspects of the assurance process (e.g., hazard and risk analysis), and then instantiate these models for concrete assurance activities for a given system. Underpinning their approach is the idea that the technical details of the system should remain in system models, which are connected to the AC as evidence, and that the AC argument should be focused on the activities that provide assurance [100]. Like Kokaly's method, they showed that Workflow+ can be used to perform change impact analysis on the AC [101].

Generating ACs at Scale. The United States Defense Advanced Research Projects Agency (DARPA) created the Automated Rapid Certification of Software (ARCOS) program to address challenges that arise while certifying and assuring complex defence technologies. A key challenge for this program was addressing assurance at enormous scales[11]. Participants in the program built solutions around ACs [102, 103, 104]. However, manually crafting an AC was not an option at this scale, so ACs were generated using libraries of AC patterns instantiated over large databases of evidence. The results from the ARCOS program demonstrate that it is possible to generate very large ACs for complex technology.

Generating ACs for Product Lines. Organizations occasionally develop critical system *product lines* where a concrete system is configured over a series of variability points. It is necessary to produce an AC with the specifics for each variant of the system, but each instance will also have much in common with the others in the product line. To reduce repetition, an abstraction of the AC can be developed called a "product line assurance case" [95]. Habli and Kelly developed an approach to managing variability in product lines using GSN patterns and GSN modules [96]. They used AC patterns to model variability points which are traced to system models. Nesic et al. used a contract-based approach to address scaling challenges for generating product-specific instances of a product line AC [105] and Murphy et al. further generalized their approach [106].

[11]To emphasize the scale, Chuck Lutz described a term in common use among his assurance colleagues that the ACs resemble "broccoli" more than "trees".

Using Generative AI to Create Arguments

The remarkable natural language processing capabilities of Large Language Models (LLMs) and Generative Artificial Intelligence (GenAI) can be used to generate ACs. Though this area is still emerging, several results suggest that this is feasible[12].

Sivakumar et al. conducted an experiment where they prompted an LLM with descriptions of a system, the argument's top-level objective, and descriptions of available evidence [107]. They compared the generated ACs with ground truth ACs and found that LLMs have a moderate capability to perform this task. Odu et al. also generated AC argument structures, but used AC argument patterns/templates to guide the LLM, which they expressed as logical predicates [108, 109]. They found that patterns helped the LLM produce "relatively good" ACs compared to a ground-truth; however, they also noted the LLMs occasionally missed relationships in the provided pattern structures. Chen et al. proposed the *Trusta* method, which uses the CAE "building blocks" to guide an LLM to perform rational argument steps when generating an argument [110]. They then check the correctness of the generated arguments via formal proof.

Despite early promise, there is reason to be cautious about using GenAI to generate ACs. In 2025, Graydon and Lehman reviewed the available results from the field and concluded that there was not enough evidence to demonstrate that LLMs were trustworthy as a tool for generating ACs for critical systems [112]. An open question remains about how to qualify an LLM for use with AC [111, 113].

2.3 The Evaluation Phase

Once development of an AC has progressed, it is common to ask: *do we have enough confidence in this argument to deploy the system?* Towards answering this question, the Evaluation phase is undertaken with the overall goal of determining if the AC's top-level claim is shown with an acceptable level of confidence. There are numerous methods for accomplishing this objective which are collectively referred to as "confidence assessment methods" (CAMs)[13]. Broadly, they can be categorized by whether they use numbers (or not) to assess confidence in an argument: qualitative, quantitative, or mixed. The remainder of this section uses this categorization to organize a

[12]This brief survey was previously published in [111].

survey of existing CAMs, and discusses their role in AC evaluation.

Remark on AC Method Phase Transitions. Despite Figure 2.1 portraying a "clean" transition from Development to Evaluation, it is common to iterate between these two phases extensively as the AC matures. In particular, the Challenge step within the Evaluation phase might generate defeaters whose rebuttal requires (re-)development of the argument or the procurement of additional evidence.

2.3.1 Qualitative Confidence Assessment Methods

Qualitative assessment methods do not rely on numerical valuations of confidence in the AC, instead, they apply specialized notations or systematic criteria to determine if an AC has satisfied its top-level claim. Five qualitative methods are described below. Of these, the dialectic reasoning method is described in some detail as it is essential to contributions in later chapters.

Challenge through Dialectic Reasoning

Like many methods used in science and engineering, ACs are prone to biases that might impact their quality or trustworthiness. In particular, ACs are prone to *confirmation bias* wherein their developers only present arguments and evidence consistent with what they already "know" to be true and alternative, possibly less flattering, perspectives are not taken into account [54]. While confirmation bias in an AC is a somewhat abstract concept, it can have real-world consequences. For example, confirmation bias was implicated in the fatal crash of the Nimrod military aircraft on September 2nd, 2006 during a "routine mission" in Afghanistan, killing all 12 crew [55]. Though a safety case was prepared for the Nimrod, it was based on previous (and implicit) safety rationale that was thought to be true by the system developers. Among the findings of the accident investigation report was:

> "*The Nimrod Safety Case process was fatally undermined by a general malaise: a widespread assumption by those involved that the Nimrod was 'safe anyway' (because it had successfully flown for 30 years) and the task of drawing up the Safety Case became essentially a paperwork and 'tickbox' exercise.*" [55, p. 10].

[13] The word "confidence" is used somewhat imprecisely here with the intent of capturing a wide range of methods. Subsequent chapters will distinguish this from related terms, like "belief".

Engaging in a dialectic process of challenge and rebuttal is a strategy for mitigating confirmation bias in ACs [20, 53, 22, 57, 114, 45]. Goodenough et al. were the first[14] to propose such a method, which is now called Eliminative Argumentation (EA) [31]. Other notations have since been extended to permit dialectic reasoning [20, 46, 32]. There are several reports of EA being applied to challenge real-world ACs [115, 53, 116, 57, 117, 118, 91].

Goodenough et al. proposed to explicitly embed sources of doubt (which they call "defeaters") in an AC's structured argument. They identified three types of doubt: rebutting defeaters challenge the validity of a claim; undermining defeaters challenge the validity of evidence; and undercutting defeaters challenge the validity of an inference rule. When preparing an AC using EA, a counterargument or counter-evidence may then be presented to rebut a defeater. If a defeater cannot be eliminated, then it is said to be "residual" and contributes to the residual doubt in the AC.

The EA notation was introduced above in Section 2.2.1 without defeaters. Now, Figure 2.7 shows the use of defeaters to challenge this argument. Defeaters are shown as red irregular octagons. `D42` is an example of a rebutting defeater that challenges its parent node, `C4`, by doubting whether the storm has sufficient energy to cross the landmass of western Vancouver Island. The defeater is rebutted by a counter-argument (`C421` and `E4211`) that appeals to measurements of the storm's energy. This counter-argument is taken as sufficient to resolve the defeater, and so it is marked with an "X"[15].

Defeater `D521` is an example of an undermining defeater that challenges the evidence provided by predictive temperature models in `E52`. A counter-argument rooted in `C5211` is presented against this defeater with its supporting sub-argument not shown. Suppose that this sub-argument is not judged as sufficient to fully resolve the doubt, so `D521` is not marked with an "X".

Defeater `D62` undercuts the inference rule `IR6` by questioning whether there might be other weather conditions that could interfere with snow arriving in Victoria, e.g., a warm front arriving from the South. No counter-argument is provided, and so this defeater is terminated by a residual node (`RES`).

[14]To the best of my knowledge, the first work on using dialectic reasoning in ACs is due to Goodenough, Weinstock, and Klein [119, 81].

[15]Strictly speaking, the "X" notation is part of the GSN notation's dialectic extension [20]. However, it is a good visual that can optionally be used to distinguish between defeaters that are resolved and those that are residual.

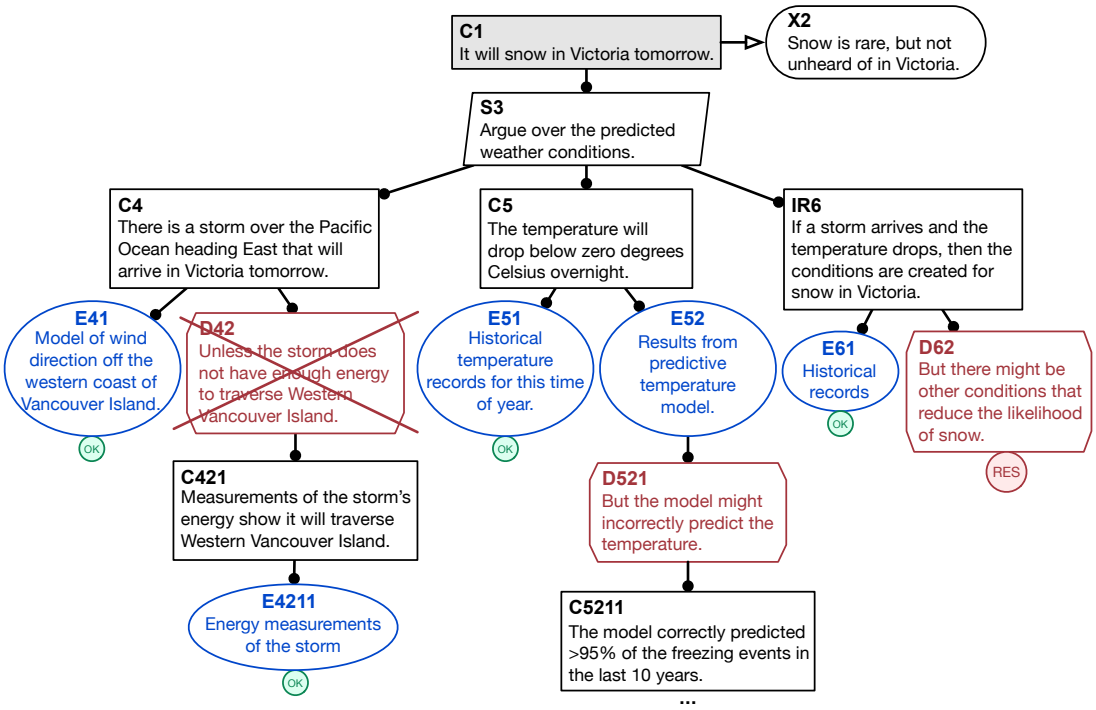


Figure 2.7: An example of using defeaters to challenge an argument about the weather in Victoria, B.C. expressed using EA.

Node and Edge Types for Defeaters. To put the defeaters in EA on a more formal footing, Figure 2.8 incorporates defeaters into the EA type graph. For clarity, the non-defeater relationships are omitted, and can be seen in Figure 2.3. The possibility for a defeater to support another defeater is an important relationship that was not shown by the example above. Sometimes a defeater can be decomposed (optionally using a Strategy node) into sub-defeaters that more precisely describe doubts. This can be useful to create additional structure an argument, especially where there are many defeaters. The situation is analogous to the decomposition of a claim into sub-claims: the child defeaters are seen as supporting their parent.

Managing Residual Doubt. When challenging an argument, residual doubts that cannot be fully resolved by counter-argument or evidence are common [53, 58]. The existence of unresolved doubt might be viewed unfavourably ("how can we release the system if we have doubts about it!?"). Imperfection, doubt, and uncertainty are simply a reality of engineering real-world systems. This is true, even if a dialectic AC is not prepared. One of the strengths of EA is that it gives permission for developers to articulate their concerns and provides a framework for reasoning about

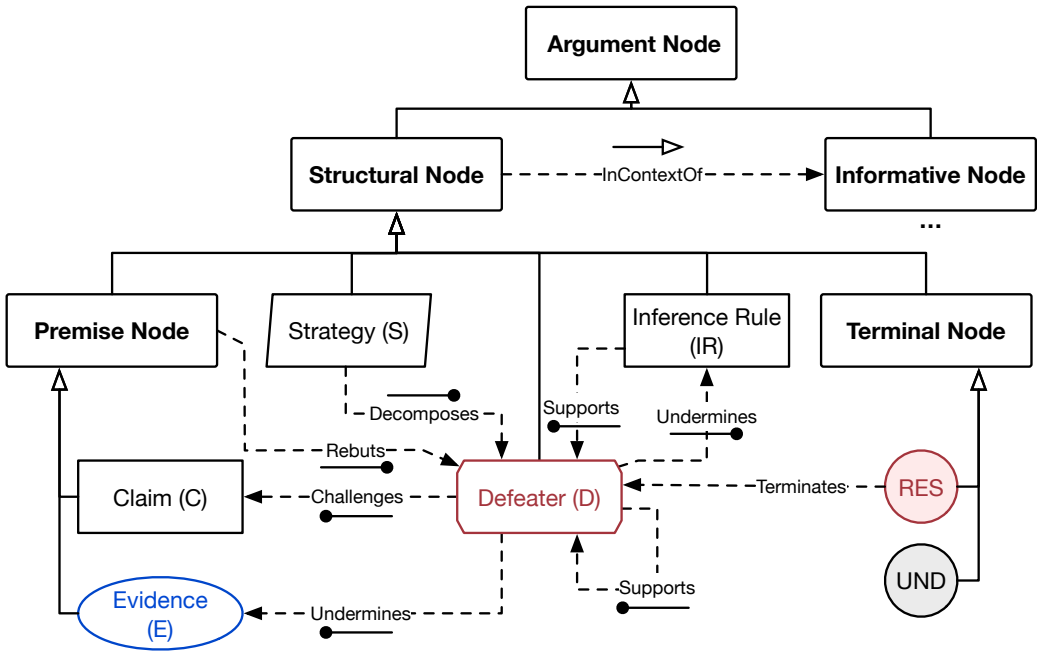


Figure 2.8: Extension of EA type graph from Figure 2.3 to include defeaters. Non-defeater relationships are not shown.

them systematically. Of course, this is not to say that we should blindly accept residual doubts. Rather, we should carefully consider residuals and apply engineering judgement about whether they are acceptable[16]. In fact, residual doubts can be used to focus discussions with decision makers and interest holders about whether a system should be deployed [53].

EA as a Confidence Assessment Method. As a confidence assessment method, EA takes the perspective that AC arguments are *defeasible*[17], meaning that it is possible that something has been omitted that might falsify the claim [31]. It follows that there must be a reason it is false. That is, a lack of confidence in a claim arises from the existence of "doubt". Without any doubt then there must be absolute confidence in the claim. Therefore, confidence in a claim can be increased if reasons to the claim are expressed and then "eliminated". This is called the "eliminative (dialectic) interpretation", and is one of at least three interpretations of defeaters in ACs, which is discussed in detail in Section 4.3.2. Using this theoretical basis, the identification and elimination of doubts can be viewed as a confidence assessment

[16]In my industrial work I have prepared and reviewed many ACs for critical systems. In at least one case, deployment of a high-risk piece of infrastructure was delayed due to the nature of the doubts we encountered while preparing the AC using EA.

method. An extension to EA called Baconian Probabilities attempts to quantify confidence using this notion [81], see Section 2.3.3.

Using GenAI to Suggest Defeaters. Using Generative AI to suggest defeaters for ACs has recently emerged as an interesting avenue of investigation [121, 122, 123, 124]. Defeaters are often discovered by brainstorming and discussion among human experts, a process which might overlook potentially important defeaters. GenAI can be used as a brainstorming aid in this process. Indeed, Viger et al. showed that approximately 25% of defeaters generated by a GenAI were considered useful challenges to samples of real-world ACs [122]. Moreover, compared to generating the "positive perspective" of the AC, the consequences of an AI suggesting an incorrect defeater are largely mitigated by the fact a counter-argument must be made against each defeater and that unfounded defeaters will be caught and removed during this process. However, using GenAI in this way is not entirely without risk, and additional work is required to show that GenAI is trustworthy for use with ACs [112, 111].

Assurance Claim Points

Hawkins et al. proposed Assurance Claim Points (ACPs) to separate an AC into two independent, but connected, argument structures [47]. The first is the main assurance argument that is typically focused on risk reduction and mitigation (sometimes called "risk argument" [45]). The second is a confidence argument that shows why the authors are confident in the rationale presented in the main argument. Hawkins et al. refer to the combination of these two argument structures as an *assurance safety argument* (respectively, security etc.).

The use of confidence arguments is motivated by the fact that plain assurance arguments can become large and difficult to understand when claims about assurance and claims about confidence are mixed. Hawkins et al. suggest that the assured safety argument approach helps to improve clarity and reduce the complexity of the core safety argument because the extra information about confidence can be examined separately, if desired [47]. Since their introduction in 2011, ACPs have since been incorporated into the GSN Community standard [20] and are also part of MISRA's GASA guidance for preparing assurance cases for automotive systems [86].

Hawkins et al. present assured safety arguments as qualitative assessments of

[17]The word "defeasible" means "*Of reasoning, a proposition, argument, etc.: capable of being overturned or disproved by further evidence; that acknowledges the fallibility of a conclusion; open to valid objection.*" [120].

uncertainty. Each ACP fragment makes a definitive statement like "there is sufficient confidence in ...". It is up to the reader of the AC to determine how much uncertainty remains after examining the supporting confidence argument. Indeed, they state that their focus is on qualitative, not quantitative, notions of confidence; though they do not preclude a quantitative approach [47].

The main safety argument and the confidence argument interface at assurance claim points[18]. These are depicted as shaded rectangles on the edges of the conventional GSN structure, as shown in Figure 2.9. Each ACP is connected to the root of a dedicated confidence argument fragment about the associated logical step in the main argument. Hawkins et al. identify three types of ACPs: *asserted solutions* (resp. evidence in EA), which focus on whether evidence encoded in a solution node is acceptable; *asserted inferences*, which focus on whether an argument step is acceptable; and *asserted contexts*, which focus on whether the information in a context node is true. Assembling all the ACP fragments together using a pattern from [47] creates the overall confidence argument.

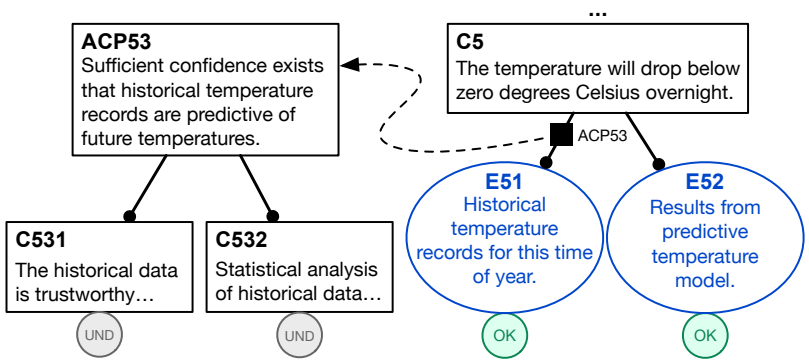


Figure 2.9: Example of a main argument and a confidence argument connected via assurance claim points.

The assured argument in Figure 2.9 contains an asserted evidence that is supported by `ACP53` containing a confidence argument about why temperature records are trustworthy. Following the structure suggested by Hawkins et al. in [47], the confidence argument for asserted evidence contains sub-claims about the trustworthiness of the evidence itself (`C531`) and the applicability of the evidence to support the parent claim (`C532`). For brevity, both sub-claims are left undeveloped (`UND`), but they would normally have their own evidence or sub-argument.

[18]It is easy to misinterpret "point" as a number like "claim X is worth 5 points", but this is incorrect. In fact, it is intended as "point on a line", or in this case, locations in the AC.

The ACP method straddles the boundary between the Development and Evaluation phases introduced in Section 2.1. On one hand, the method was conceived as a means of improving the clarity of arguments during their development and as a means of *supporting* the main argument. On the other hand, creating ACPs is a (qualitative) confidence assessment method, which is considered part of the evaluation phase. In the end, it is discussed here as part of the Evaluation phase because, in a recent paper on the overall AC method, Hawkins positioned confidence arguments to address challenges raised against an AC [45], which is part of the Evaluation phase.

Review and Interrogation of ACs

Peer review is a cornerstone of quality in engineering and science. In keeping with this practice, ACs are also regularly reviewed as part of quality management processes. An additional level of rigour can be achieved by engaging *independent assessors* to review an AC and determine if its top-level claim is satisfied with an acceptable level of confidence. For a critical system, it is common to require an independent assessment as part of the regulatory process. Further, multiple "layers" of reviews and assessments might be required. For example, ISO 26262 requires reviews of engineering work products; then a dedicated "confirmation review" of the safety case by an independent reviewer where the level of independence is increased with the criticality of the system; and finally an overall assessment for compliance with the standard is performed by an independent assessor, which includes a review of the safety case [13].

There is no single method that is recommended for reviewing an AC. Though, as with any technical review, it is necessary to engage more deeply with the material than simply checking for typos. A common practice is to apply criteria systematically to the AC, either considering the AC as a whole or *interrogating* each argument step in isolation. Two approaches for systematically reviewing an AC are described below.

The iTest Method. Holloway and Wasson proposed the iTest method for qualitatively evaluating arguments [49]. While they have described their method in terms of the Friendly Argument Notation, the underlying concepts are broadly applicable to other notations and approaches for expressing arguments. The iTest method is performed iteratively, where compound arguments have their atomic sub-arguments isolated and interrogated. Each atomic argument is assessed using the SPRY mnemonic as a guide: "is the Syntax proper?", "are the Premises acceptable?", "is the Reason-

ing acceptable?", and "is saying Yes to the conclusion justified?" [49]. iTest has been used for whether aerospace systems possess desirable "over-arching properties" [125].

Chowdhury's Evaluation Criteria. Chowdhury et al. defined two sets of criteria that can be used to systematically review an AC [126, 127]. They describe each criterion from the perspective of an AC developer and an external reviewer. Their dual-perspective description helps to ensure that each party understands how to best satisfy each criterion.

The first set of criteria are focused on the well-formedness of the AC's structure and notation: *syntax*, *traceability*, *robustness*, *understandability*, and *efficiency*. As an example, the developer perspective for the *traceability* criterion says that an AC developer should maintain traceability to artifacts that support the AC, even during maintenance, and the external reviewer perspective says that the reviewer should confirm that links to artifacts exist, and they are easy to identify and trace. The second set of criteria are focused on the content of the AC: *convincing basis*, *rigour in argument*, *quality of hazard analysis* (for a safety case), *argument completeness*, *repeated arguments*, *ALARP*, and *confidence*. As an example, the *repeated arguments* criterion aims to identify and mitigate the impact of repeated arguments in the AC.

Safety Assurance Levels

Weaver et al. provided an early example of qualitative reasoning about confidence using "Safety Assurance Levels" (SALs) [128]. Their approach is inspired by the established notion of a Safety Integrity Level (SIL) or Design Assurance Level (DAL) that is used by many technical standards as part of a level of rigour approach to risk management. In Weaver et al.'s method, the AC's top-level claim is assigned a SAL, which is then partitioned among its child claims according to decomposition rules. Partitioning continues recursively until leaf nodes in the argument are assigned SALs indicating the level of rigour that must be applied in procuring evidence to support the argument. Interestingly, unlike many other CAMs that reason about confidence in a "bottom-up" manner, Weaver et al.'s method approaches confidence from the top-down. That is, it is a prescriptive approach to AC confidence, where the necessary level of rigour in the evidence is determined based on the AC's argument structure, and then evidence is assessed to determine if it satisfies this requirement.

Three-Valued Logic and Strength of Knowledge Criteria

A recent proposal by Hafver et al. suggested using three-valued logic (true, uncertain, false) to reason about confidence in AC claims [129, 130]. They apply several "strength of knowledge" criteria to decide on the level of confidence in the evidence at the leaves of an AC's argument. For example, part of their *Method* criterion is: "the methods used to obtain the evidence are recognized and have been applied correctly according to best practices..."[130, Table 1]. Then they use truth tables from their three-valued logic to propagate their assessments through the argument. Their proposal is motivated by challenges in quantitative CAMs, such as those raised by Graydon and Holloway [80].

2.3.2 Quantitative Confidence Assessment Methods

Compared to qualitative methods, quantitative methods represent confidence in the AC numerically (e.g., on the interval $[0, 1]$.) Numerous quantitative methods have been proposed. At a high-level, these methods usually involve three steps: 1) the user assigns confidence values for each leaf node in the argument; 2) the user assigns parameters to each argument step describing how confidence in the children impacts the parent; and 3) using an algorithm to recursively calculate confidence at each argument step based on the user's assignments. The available methods can be categorized according to their underlying mathematical theory. The remainder of this subsection briefly introduces each theory and then, drawing on existing methods, discusses its application to AC confidence assessment.

Confidence Assessment with Probability Theory

Probability is a well known theory for describing uncertainty that models the likelihood of an event occurring on the interval $[0, 1]$. For AC confidence assessments, the probability measures correspond to a subjective measure of belief that a claim in the argument is true. Bayesian Networks (BNs) are a useful technique for applying probabilistic reasoning problems that involve many interconnected outcomes.

BNs were first introduced by Judea Pearl in 1988 [131] and have been used in a range of fields, ranging from artificial intelligence, medicine, economics, and social science. A BN encodes an analyst's understanding of the world (outcomes and their relationships) as a directed acyclic graph where nodes are outcomes and directed

edges ("arrows") describe relationships where the target of the arrow indicates that an outcome is influenced by the outcome at the source of the arrow. The BN is accompanied by conditional probabilities (typically expressed as tables) that indicate how specific outcomes influence each other. An important use case of BNs is to perform "what-if?" analyses to determine how changes in the likelihood of one outcome will propagate through the network [132, 133].

Given the wide use of BNs in other areas of science and engineering, it is not surprising that they have been used as the basis for several AC confidence assessment methods. Though the details differ between methods, the general strategy is to derive a BBN that mirrors the structure of the AC argument and then to elicit expert opinion on the leaves of the argument in terms of a subjective probability or degree of belief about the trustworthiness of a claim or supporting evidence. These assessments are used to compute the probability (as degree of belief) in the top-level claim of the argument using the rules of conditional probability according to the BN's structure.

Early Work on BNs for Assurance. Early examples of using probability theory to reason about safety assurance include work by Fenton et al. [134], Littlewood et al. [135], and Guo [136]. For example, in 2003 Guo represented confidence in whether the "overall safety requirement" for a system had been achieved using a BN that mirrored the structure of compliance requirements from an industrial standard, such as IEC 61508. These early works identified an important advantage of the BN approach, and quantitative assessments more broadly: the ability to capture sources of uncertainty beyond strict failure modes of a system. For example, the skill level of software developers or impact of verification methods could be (subjectively) modelled. Fenton also recognized the value of BNs in explicating structured assurance arguments, which was an emerging idea at the time [132, 27, 28].

The BBN Method for AC Confidence Assessment. Building on the earlier works, Hobbs and Lloyd described the application of Bayesian Belief Network (BBN) to assess confidence in a real-world AC [48]. In their paper, they described a method for constructing the BBN and using it to evaluate belief in an argument. Their method is taken as a prototypical example of the BBN method, and since later chapters will build upon this method, it is described in detail[19]. Related BBN-based methods are then discussed.

Let $\Pr(c_i) \in [0, 1]$ represent a measure of subjective belief in a claim in an argu-

[19]This description is an abbreviated version of the description from [84].

ment, where 1 means the claim is believed to be true and 0 means it is believed to be false. Given a single argument step, $c_0 \multimap c_1, ..., c_n$, the computation has two parts: 1) compute the conditional probability table (CPT) describing the relationship between the parent and child beliefs, and 2) use the CPT to propagate belief from the child nodes to the parent.

Prior to computing the CPT, users must specify several parameters for each argument step that collectively describe the relationship between the parent and its children. The parameters are:

- **Logical Combinator** - The operator describing how the children logically combine to support the parent, either `AND` or `OR`.
- **Leakage Parameter** - The leakage, $k_i \in [0, 1]$, captures the possibility that belief in the parent is not entirely determined by the belief of the children. The semantics differ based on the combinator. For `AND`, the leakage, k_0, gives the belief that the parent is false, even if all children were true. For `OR`, the leakage gives the belief that the parent is true, even if all children are false.
- **Link Probability** - Each child in the argument step has a corresponding link probability, $w_i \in [0, 1]$, that describes the impact of the child on the parent's belief. The semantics change based on the combinator parameter. For `AND`, the degree of belief in the parent is $1 - w_i$, if only child c_i is false and all other children are true[20]. For `OR`, the degree of the belief in the parent is w_i, if only child c_i is true and all other children are false[21].

The above parameters are used to compute a NoisyAND or NoisyOR (respectively) CPT for the argument step. The CPT contains the belief in the parent over all possible (Boolean) values of the children. Each row in the table corresponds to a possible assignment of belief to the children with $x_i \in \{0, 1\}$ giving the Boolean value of that node in the row. For NoisyAND and NoisyOR, the two computations are:

$$\Pr(c_0|c_1, ..., c_n) = (1 - k_i) \prod_{i=1}^{n} (1 - w_i)^{1-x_i} \tag{2.1}$$

[21]Some mental gymnastics might be required when selecting link probabilities. Chris Hobbs provided me with a nice template sentence for `AND`: "*If everything else was perfect, but [CLAIM is false], how much would we reduce our belief in the parent claim?*".

[21]Similarly: "*If everything else was hopeless, but [CLAIM is true], how much would our belief increase in the parent claim?*"

$$\Pr(c_0|c_1,...,c_n) = 1 - \left[(1-k_i)\prod_{i=1}^{n}(1-w_i)^{x_i}\right] \tag{2.2}$$

Given belief assignments for the children in an argument step, the CPT's conditional probabilities are then used to compute belief in the parent using the standard propagation formula for a BN:

$$\Pr(c_0) = \sum_{x_i \in \{0,1\}} \Pr(c_0|c_1 = x_1,...,c_n = x_n) \cdot \Pr(c_1) \cdot \ldots \cdot \Pr(c_n) \tag{2.3}$$

As an example, consider the argument step in Figure 2.10 that uses the `AND` combinator ($k_5 = 0.2$). Suppose our belief that the temperature will drop below freezing, c_5, is weighted more heavily on predictive modelling results, e_{52} than on historical records, e_{51}. If we had perfect historical records, but did not have predictive modelling results, then our belief in the parent would drop by $w_{52} = 0.8$. Conversely, if we had perfect modelling results, but no historical records, then our belief in the parent would only drop by $w_{51} = 0.3$. Using these parameters, the NoisyAND CPT can be constructed, as shown in Table 2.1. Then supposing, the belief assignments at the leaves are $\Pr(e_{51}) = 0.7$ and $\Pr(e_{52}) = 0.9$, the belief in `C5` calculated to be $\Pr(c_5) = 0.67$.

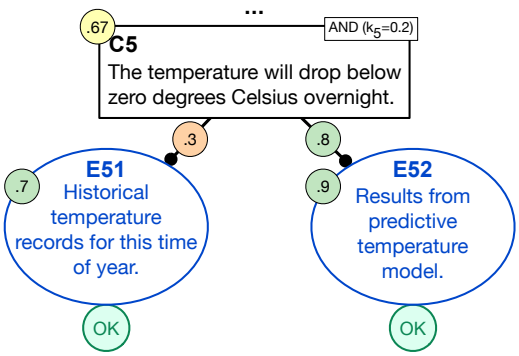


Figure 2.10: Example of the BBN method applied to a single argument step.

Related Works using BNs for Confidence Assessment. In addition to Hobbs and Lloyd, a number of authors have developed BN-based methods that take advantage of an AC's structured argument (usually expressed using GSN). Denney et al. described a method that models belief using probability distributions over an ordinal scale with five confidence values: *very low*, *low*, *medium*, *high*, and *very high* [56]. In Denney et al.'s construction, each argument step has a probability distribution

$\Pr(c_{51})$	$\Pr(e_{52})$	$\Pr(e_5)$ (AND)
0	0	0.11
0	1	0.56
1	0	0.16
1	1	0.80

Table 2.1: Sample NoisyAND CPT with $w_{51} = 0.3$, $w_{52} = 0.8$, and $k_5 = 0.2$

describing the overall confidence in the truth of the step's parent claim. Zhao et al.'s method use a meta-model to define transformations between argument steps expressed using GSN or CAE into an argument fragment in Toulmin's form, and then they use a second transformation to generate a BN-based on the Toulmin fragment [137]. Nešić et al. defined an abstract model of structured argument using model theory, and then used that model to formalize a transformation from a structured argument to a BN [138]. An interesting part of their transformation is a procedure for constructing the BN's CPT based on three different types of argument steps. Rushby suggested using BNs as a means of performing "what-if?" analyses on argument structures to determine their sensitivity to individual claims or pieces of evidence [139]. Oh et al. developed the ARACHNE method to formalize the claims in an AC as a series of stochastic propositional logic expressions they call "contracts" [140]. Then they used a BN to reason about the overall confidence in the argument while also accounting for the satisfiability of logical expressions [140]. Unlike other authors, Asaadi et al. used Dynamic Bayesian Networks (DBNs) to describe confidence in a claim, but not to propagate confidence through the argument. Instead, they use the DBNs to produce "assurance measures" (as in Section 2.2.3) that measure critical properties of a system. They link these measures to leaves of a structured argument to produce a DAC.

Confidence Assessment with Dempster-Shafer Theory

Dempster-Shafer Theory (DST), also called "evidence theory" generalizes probability theory by modelling events using two complementary measures: *belief* and *plausibility* [141, 142, 143]. There are at least two differences between DST and probability theory. First, probability theory permits two outcomes for assertions: true or false. Ignorance in probability is modelled by the uniform distribution (all outcomes are equally likely). However, in DST one can describe an assertion as being true (high belief), false (low

plausibility), or uncertain (plausible but not believable). Second, probability can be interpreted in an empirical sense (sometimes called the "Frequentist" perspective) where the probability of an event is based on experimental outcomes [142]. For example, after an infinite number of tosses of a fair coin the probability of "heads" is said to be 0.50. In contrast, DST takes a strictly subjective perspective without requiring empirical grounding of the belief or plausibility of outcomes. More detail on DST is provided in Section 5.2.

Several CAMs based on DST have been proposed. Among them, Idmessaoud's is the most recent, builds on other works [79, 144], and is representative of this type of method [50]. It is described in some detail and then other formulations are briefly reviewed.

The DST Method for AC Confidence Assessment. Additional measures can be created based on the DST's core belief and plausibility measures. To create a CAM, Idmessaoud et al. derive two measures: *decision* and *confidence* [50]. The decision measure gives the degree of acceptability of the claim, from rejectable to fully acceptable. The confidence measure describes the degree of conviction or certainty in the decision. Claims are represented by pairs of measures: (d, c) for $d \in \{d_1, ..., d_5\}$ with d_1 indicating the claim is rejectable and d_5 meaning the claim is acceptable, and $c \in \{c_1, ..., c_5\}$ with c_1 meaning a lack of confidence and c_5 meaning complete confidence. These two measures are bound together using *Josang's Constraint* which says that extreme decision values entail high confidence, and that as decisions become less extreme, confidence may (optionally) decrease. For instance, the pair (d_5, c_1) is forbidden because it indicates a claim is fully acceptable, but with no confidence.

To assess an argument, each leaf node is assigned a decision and confidence pair. At each argument step several parameters are defined to describe the impact of the children on the parent. For each premise-typed child in an argument step, two parameters are assigned: the *direct* parameter, describing how the truth of the child impacts the parent, and the *reverse* parameter, describing how a false child impacts the parent. Additionally, for some propagation operators, a direct and reverse parameter for the whole argument step is required. Four propagation operators are defined: S-Arg (simple, single child), D-Arg (disjunctive), C-Arg (conjunctive), and H-Arg (hybrid). Of these, the H-Arg is the most interesting, corresponding to the case "where each premise supports the conclusion to some extent, but their conjunction does it to a larger extent" [50]. In the limit cases, the H-Arg becomes either the D-Arg or C-Arg.

DST-based methods typically use a graphical input and output where the user indicates their decision and confidence levels on a two-dimensional grid shown e.g., the grid in Figure 2.11. The output of the assessment is shown to the user in the same format. The user selects the propagation operator for each argument as well as specifying the direct and reverse parameters numerically.

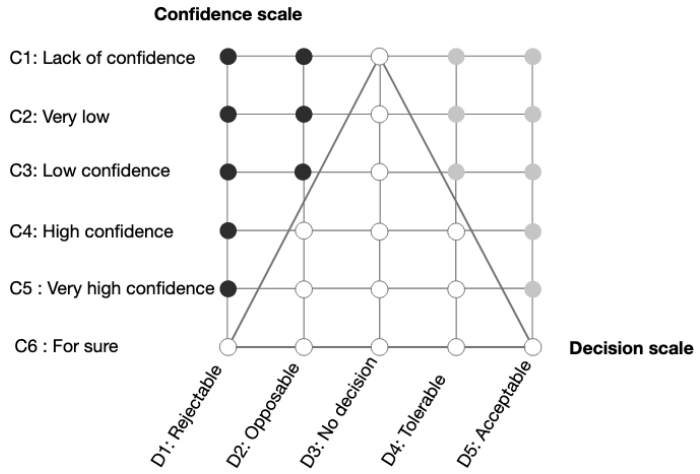


Figure 2.11: The decision and confidence grid used by DST-based confidence assessment methods, from [50]. Josang's constraint overlaid as a triangle.

Related Works using DST for Confidence Assessment. Cyra and Gorski were the first to apply DST to assess AC arguments [145, 146, 79]. The DST method described above incorporates many of their ideas, including the basic propagation logic, ordinal decision and confidence scales, and use of Josang's constraint. In their initial work they modelled arguments with Toulmin's form. Cyra and Gorski conducted a survey to determine the mapping between qualitative scales for decision and confidence and the corresponding values used as input to their DST confidence propagation formulas [79]. Participants assessed several argument steps on their two-dimensional scale. In aggregation, the survey participants produced non-linear scales (e.g., spacing between high and very high confidence is not the same as the spacing between very high and for sure confidence).

Several authors have built on Cyra and Gorski's initial method. Ayoub et al. extended the method to handle GSN-based arguments and considered slightly different measures for evidence (*sufficiency* and *insufficiency*) [147]. Wang et al. added new argument step types and considered *appropriateness* and *trustworthiness* measures [144]. Since the DST method is sensitive to the choice of parameters, Wang et al. de-

veloped a parameter calibration method that asks experts to assess simple argument fragments [148]. Wang et al. also applied their method to a rail safety case derived from the rail signalling functional safety standards (the EN 5012x series) [149].

Finally, Guiochet et al. modelled an argument's confidence using DST and then formulated a BN encoding the argument structure [150]. Of particular interest is their proposal to conduct sensitivity analyses on their model (depicted as a "tornado diagram") to determine how sensitive the argument is to change in specific elements.

Confidence Assessment with Subjective Logic

Subjective logic uses tuples (called "opinions") of probability measures to reason about belief and confidence in the truth of propositions [151]. The parameters in these tuples collectively describe a Beta distribution that represents the confidence in an AC's claim [152, 153, 154]. An important feature of subjective logic is the variety of operators available for manipulating opinions, which authors have used to describe confidence propagation for AC argument steps. For instance, Herd et al. use subjective logic's *conditional deduction* operator to combine the belief of children in an argument step.

Confidence Assessment with Achievement Weights

For assessing compliance with security requirements during system adaptation, Jahan et al. developed a method to compute a "satisficing level" ($L \in [0, 1]$) for each security requirement in an AC [155, 69]. They model each security requirement as a *softgoal*, and its supporting claims from the AC, as a Softgoal Interdependency Graph (SIG). As a softgoal, each requirement is satisfied if it is achieved "well enough", but not necessarily perfectly [69]. They compute *achievement weights* ($a_w \in [0, 1]$) for each claim in the AC. At the leaves (i.e., for evidence in the AC) achievement weights are calculated based on pre-defined *impact factors* ($I \in [0, 1]$) that correspond to security design criteria (e.g., how are audit trails managed by the system). Achievement weights for a parent claim are computed as the average of child achievement weights; no other propagation operators are defined. The satisficing level for a requirement is computed as the average of the supporting claim's achievement weights summed with the achievement weights of neighbouring goals in the SIG (accounting for overlap).

From a theoretical perspective, Jahan et al.'s proposal to use achievement weights and satisficing levels does not appear to be grounded in any theory of uncertainty

or imperfect knowledge, so the validity of their construction is not clear. Moreover, the method only permits averaging of achievement weights; no other propagation operators are defined, which limits the types of argument steps that can be modelled.

From a methodology perspective, Jahan et al.'s proposal is focused on measuring the satisfaction of security requirements that appear in an AC. However, ACs are not only a means of modelling requirement satisfaction, other types of claims might be necessary for security assurance. For example, it is necessary to also argue that the security requirements, if satisfied, adequately mitigate threats. Unlike other quantitative CAMs described above, Jahan et al.'s method is focused on computing a satisficing level of requirements, not computing an overall level of confidence in an AC.

2.3.3 Mixed Confidence Assessment Methods

Some CAMs combine aspects of both quantitative and qualitative methods, which are referred to as "mixed". Two such methods are summarized below.

Assurance 2.0

Bloomfield and Rushby created the Assurance 2.0 framework [156, 46]. This framework expresses ACs using the CAE notation. It considers ACs from three perspectives: positive, negative, and residual.

From the positive perspective, the argument is viewed in terms of how the available evidence and argument structure collectively support the top-level claim. They consider an AC to be *valid* if every step in the argument is deductive[22] (i.e., the premises are sufficient to imply the conclusion) and there are no unresolved doubts (defeaters) in the step. They consider an argument *sound* if all steps are judged by an expert to be justified and all provided evidence is credible. They introduce the notion of a confirmation measure which is a subjective probability of a claim's truth with and without the supporting evidence; to be credible, evidence must sufficiently influence belief in the claim. Finally, once an AC is determined to be sound, they evaluate confidence using a (subjective) probabilistic interpretation of confidence where confidence is propagated through the argument using conditional probabilities and logical connectives. Assurance 2.0 defines five argument steps (in CAE) that may be used to build an AC's argument: *evidence incorporation*, *decomposition*, *substitution*, *concretion*, and *calculation*. They claim that limiting an argument's structure to these

five steps increases deductivity and assists in the assessment of argument validity.

From the negative perspective, Bloomfield and Rushby permit the expression of doubt as defeaters. Only once all doubts have been resolved is the argument said to be *indefeasible*, a criterion for validity.

Finally, in the residual perspective, defeaters that cannot be resolved are marked as *residual* and must be analyzed. They propose a qualitative scale for evaluating residual defeaters and suggest that only residuals in the lower criticality categories be accepted.

Baconian Probabilities

EA was introduced as a qualitative method above. However, a quantitative extension using Baconian probabilities exists for the method [81, 119]. In the philosophy of science, a Baconian Probability (a term due to Cohen based on Bacon's scientific method [82]) is an ordinal measure that may be used to rank competing hypotheses based on the number of "tests" passed out of a total number of tests. Typically, this is denoted as $i|n$ where i is the number of successful "tests" and n is the total number of tests. In EA the confidence of an argument may be similarly expressed where i denotes the number of residual doubts and n denotes the total number of doubts. Goodenough et al. are careful to point out that a Baconian Probability is not the same as a Pascalian probability (i.e., a real number between 0 and 1 denoting a frequency or likelihood) [119]. That is, 2|4 cannot be reduced to 1|2 nor does it represent a likelihood of 0.5; instead, it should be interpreted as "the hypothesis has withstood two out of four potential reasons for doubting its truth" [119].

The utility of Baconian Probabilities in EA is uncertain. Graydon challenged it on theoretical grounds by noting that, because Baconian Probabilities are ordinal measures, they cannot be manipulated arithmetically to propagate them through an EA argument structure [82]. Graydon also questioned their utility from a practical perspective: because they are ordinal measures, they can be used for ranking but cannot be used to evaluate the "distance" between ranks [82, 53].

[22] Bloomfield and Rushby do not use any logical formalism to evaluate deductively and instead refer to "Natural Language Deductivism" (NLD) as a threshold for deductivism.

2.4 The Deployment Phase

Once an AC has been evaluated, and determined to be acceptable, a decision must be made to deploy the underlying system. In the Deployment phase, the objective is two-fold: 1) *monitor* the system to determine if the AC remains valid, and 2) *respond* if the AC is determined to be invalid.

In a process of *continuous assurance* (e.g., the *DevSafeOps* methodology [61]), the transition from Evaluation to Deployment aligns with the conventional design-time vs. run-time split. During design-time, the AC is developed and evaluated. Then, at run-time the indicators are used to confirm the AC remains valid. If required, the response is usually to (re-)initiate Development to change the underlying system and the AC. A transition back to Development might also be triggered by planned changes to the system's design (e.g., the next version of the system or product).

However, it is also possible that the transition from the Evaluate to the Deployment phases occur independently of the transition from design-time to run-time. As mentioned in Section 2.1, if the AC method is embedded within a self-adaptive system, possibly as part of the MAPE-K reference architecture sketched in Figure 2.12 [157, 67], the Development, Evaluation, and Deployment phases might be part of a perpetual assurance process that is carried out entirely at run-time [68]. Using MAPE-K, the (dynamic) AC is viewed as a run-time model of assurance within the system's *Knowledge* base ("K" in MAPE-K), and indicators are used to *Monitor* the validity of the AC. If changes are required then a response is prepared by *Analyzing* the situation, *Planning* an adaptation, and *Executing* it [67]. Some aspects of this (e.g., planning and executing) blend into Development phase activities.

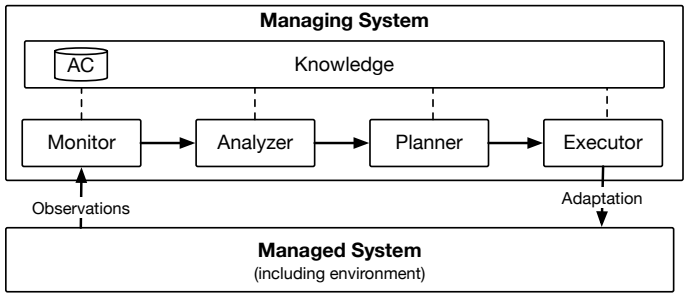


Figure 2.12: The MAPE-K reference architecture for a self-adaptive system.

2.4.1 Monitoring AC Validity with Indicators

Monitoring to confirm the AC's validity is mainly achieved through indicators. As discussed in Section 2.2.3, indicators instrument the AC with "live" data from the environment and can be used to detect invalid claims. Some methods for monitoring the AC via indicators are described below.

The AC-ROS Framework.

Cheng et al. developed the AC-ROS the framework for assurance of self-adaptive systems [158]. In their approach, the AC is part of the Knowledge within a MAPE-K architecture. The entirety of the AC is developed at design-time but may include optional modules that can be enabled (or disabled) at run-time based on the current configuration of the robot. They use 'utility functions' (embedded in GSN solution nodes) to determine whether the current operational conditions are consistent with the expectations of the AC. Their utility functions are similar to the notion of indicators described above, but use boolean functions over the monitored variables. When a utility function evaluates to false, the Monitor triggers a response via the rest of the MAPE-K architecture. Cheng et al. demonstrated the concept by implementing their framework atop the Robot Operating System (ROS) middleware to create a self-adaptive system.

Dynamic Assurance Cases for Trusted Autonomy.

Asaadi et al. used Dynamic Bayesian Networks to incorporate assurance measures into a DAC, providing run-time monitoring of the claims in an AC [63, 159]. Their approach blends quantitative assurance measures with qualitative aspects of an argument expressed using GSN, but does not generate an overall confidence assessment. Instead, the quantification of uncertainty is limited to specific nodes within a GSN argument structure. Building on earlier ideas from Denney et al. [56], they describe a reference architecture that uses the DAC to decide whether to adapt an autonomous system. They demonstrate their method on a learning-enabled autonomous runway taxiing application for unmanned aerial vehicles (UAVs).

Using Indicators to Validate ACs for Aviation.

Ferrel and Andegg propose instrumenting an AC developed for safety-critical autonomous aviation systems with Safety Performance Indicators (SPIs) to validate the AC's claims based on data from real-world operations [160]. They view this as part of a larger Safety Management System for autonomous systems. As an example, they propose an indicator based on geo-fence violations of an autonomous aerial vehicle (i.e., "drone").

2.4.2 Responding to Invalid Claims in an AC

Once monitoring has determined that one or more claims in AC is invalid, a response is required. This might be as simple as ceasing operations. However, if availability is important, it might be necessary to *change* the system, possibly via run-time adaptation or to resume design-time activities as discussed above. Methods for orchestrating this response are presented below.

The ENTRUST Method

Calinescu et al. created the ENgineering TRUstworthy Self-adaptive SofTware (ENTRUST) method for assuring self-adaptive systems [75]. ENTRUST divides the creation of an AC into two phases: design-time and run-time. At design-time, analysts prepare a partial AC that includes both i) design-time verification evidence that, for instance, demonstrates that the managing system satisfies its allocated requirements; and ii) placeholders for verification evidence that will be generated at run-time as the system adapts. At run-time the managing system performs automated verification tasks to show that a new configuration satisfies the allocated requirements. The results of run-time verification are provided to complete the instantiation of the AC for each configuration of the SAS. This means that, at any point in time, an analyst may inspect the AC to understand how the current configuration is assured. ENTRUST does not explicitly use an instrumented AC to monitor the environment, and instead focuses on updating the AC's evidence.

ENTRUST includes an AC pattern with two main branches. The first branch argues over the allocated system requirements to show that they are satisfied by the current configuration. Each requirement is supported by formal verification evidence that is automatically generated by the managing system at run-time for the

configuration. The second branch argues that the managing system performing the adaptation will not exhibit erroneous behaviour and addresses aspects such as failure mode analysis, engineering process, and reliability of tools used for verification. Since the managing system itself does not adapt, the evidence in this second branch is generated at design-time.

Self-Adaptation for Security ACs

Jahan et al. consider how a self-adaptive system can satisfy security requirements, when adaptations are performed [161, 162]. This work culminated in Jahan's PhD thesis [69]. In their method, a dynamic security case argues that a system's security requirements are satisfied. The case's structure is adapted at run-time via adaptation operations. To reason about the impact of adaptation on security requirements, Jahan et al. compute satisficing levels as described in Section 2.3.2. Lower satisficing levels imply increased risk of non-compliance with a security requirement. They apply a similar approach for the satisficement of functional requirements. In their proposed architecture, satisficement levels are provided to a coordinator that decides whether an adaptation is necessary. The coordinator aims to always satisfy functional requirements and make a "best effort" to comply with security requirements.

Dynamic Modular ACs

Mirzaei et al. argue that safety cases for adaptive systems-of-systems must be both modular and dynamic. They suggest that modular safety cases allow components of a self-adaptive system, each with its supporting safety case, to be interchanged at run-time. Mirzaei et al. suggest combining these two concepts to create Dynamic Modular Safety Cases (DMSCs) for adaptive systems-of-systems. Thomas et al. applied this concept to systems-of-systems that spontaneously re-configure themselves using a service-oriented architecture [74]. They define safety case modules for each service that could be incorporated into the system. Each module includes a contract consisting of "Assume" and "Guarantee" assertions over the case that were specified at design-time by a human expert. At run-time, they automatically compose a safety case from the available modules based on the current system configuration and verify that the contracts between modules are satisfied.

Conditional Safety Certifications (ConSerts)

Schneider et al. considered the problem of assuring "open" adaptive systems, where new components can be added or removed from the system dynamically [163, 164], with the work culminating in Schneider's dissertation [72]. They created the notion of a Conditional Safety Certificates (ConSerts). Each component has a corresponding ConSert (established at design-time) that provides both "demands" (required for the component to operate) and "guarantees" (provided by the component). At run-time, as the system adapts and components are added/removed, a ConSert tree is maintained that shows how the demands and guarantees are jointly met. Their ConSert tree is not unlike a dynamic safety case, except that their approach is narrowly focused on safety integrity levels (SILs), as defined by IEC 61508 [165]. Using ConSerts they estimate the probability of system failure for a candidate configuration.

The ReASSURE Framework

Murphy et al. proposed the ReASSURE framework that uses the notion of a dynamic product line to maximize re-use of an AC during system evolution, such as when the self-adaptive system evolves [106]. They compose a dynamic AC that includes variation points that enable assurance about system adaptations during evolution. Component contracts help to manage the composition of ACs for diverse components. At the time of writing, ReASSURE is still under development.

2.5 Chapter Summary

To introduce foundational concepts, this chapter described the overall "assurance case method" as a collection of notations and methods that are applied during the Development, Evaluation, or Deployment of an AC. Core concepts that subsequent chapters build upon include: dialectic reasoning using the Eliminative Argumentation (EA) notation, AC confidence assessment using qualitative and quantitative methods, instrumenting an AC with indicators to enable monitoring of claims for on-going validity as part of a continuous or perpetual assurance process.

Absent in the collective work forming the Assurance Case Method is a method for assessing the overall confidence in a dynamic assurance case using a combination of static (human) assessments, live data from indicators, and the availability of evidence. To close this gap, the subsequent chapters in this dissertation develop such a method.

Chapter 3
The Practitioner Perspective

An important part of the overall research goal for this dissertation is to develop a CAM that considers the needs of its end-users, i.e., AC practitioners. While many CAMs have been proposed by researchers, little is known about their use during the development and management of real-world ACs. In part, this gap in knowledge is driven by the nature of the information captured in ACs. They almost always contain sensitive or confidential information (e.g., trade secrets or residual sources of risk) that many organizations are unwilling or unable to disclose in public venues, such as scientific conferences or journals. There are only a handful of comprehensive case studies of ACs for real systems, and even fewer reports of using CAMs. As a result, researchers are left with a limited data set to use as a basis for developing methodologies. Before designing a new CAM, it is necessary to understand the current state of practice for AC confidence assessment. Since this cannot be easily established by analyzing published case studies, the alternative is to ask practitioners about how they gain confidence in ACs.

This chapter describes an interview study of AC practitioners aimed at developing the "practitioner perspective" on AC confidence assessment. After describing the methods and results, the study's findings are discussed in relation to similar empirical studies. Finally, the results from the study are synthesized to produce the "Practitioner Perspective", which is used as input to a requirement specification activity in Chapter 4. The empirical research methods and results described in this chapter have been published [52].

3.1 Study Objectives

Towards developing the practitioner perspective on AC confidence assessment, this interview study had four objectives:

1. **Usage** - Determine if practitioners use CAMs in their practice.
2. **Methods** - Determine which CAMs are used by practitioners when developing ACs for real-world projects.
3. **Opinions** - Elicit practitioner opinions of methods, including those they might not have applied in practice.
4. **Challenges & Barriers** - Identify challenges that practitioners have encountered in the application of CAMs, and barriers (real or perceived) to adoption of CAMs in practice.

3.2 Study Methods

This study used a grounded theory methodology aimed at developing the practitioner perspective on AC confidence assessment [83]. Data was collected using structured interviews with AC practitioners and a follow-up email questionnaire. Open-coding analysis was performed on the interview transcripts and emailed responses. The study used a cross-sectional approach (i.e., a "snapshot" in time), and was designed based on the guidance in [166, 167, 168]. Since this study involved human subjects, the methods were reviewed by the University of Victoria's research ethics office.

3.2.1 Recruitment

Recruitment used a "convenience" sample from my professional networks. Invitations were sent via email to persons with known industrial experience with ACs (i.e., practitioners). Recruitment was targeted to capture a wide range of experience from different industries. To be included in the study, a participant must have: (1) had experience contributing to one or more real-world ACs; (2) been a working professional or recently retired; (3) been at least 18 years of age; and (4) been professionally fluent in English. Invited participants self-applied the inclusion criteria as part of the initial

invitation and, if they proceeded to an interview, were screened again by the interviewer with survey questions (see Questions #1 and #3 in Appendix A). Participants that were interviewed, but who did not satisfy the inclusion criteria as determined by screening questions, completed the interview but their interview transcript was not analysed.

3.2.2 Data Collection

Data collection proceeded in two phases. The first phase used structured interviews, and was the primary mode of data collection. The second phase used an email questionnaire to gather additional information on specific topics of interest.

Phase 1: Interviews

Structured interviews with synchronous discussion were used as the primary means of data gathering because they allowed for in-depth exploration of topics resulting in richer data. Interviews used a predetermined set of questions (see Appendix A). In response to each question, the discussion was permitted to flow naturally allowing the interviewer to inquire further about specific topics. For consistency, all interviews were performed by the same interviewer. A pilot interview was held with one participant to evaluate and improve the questionnaire used for the study.

Interviews were held remotely, using the Zoom video conferencing service provided by our institution. Zoom was configured to save a textual transcript of the interview. Audio and video recordings were not stored. After each interview the transcript was de-identified such that the participant's name was replaced with a unique identifier (e.g., "John Smith" became "P02") and organizations were replaced with placeholders (e.g., "Acme Corporation" became "EMPLOYER").

Phase 2: Email Questionnaire

After the completion of interviews, and subsequent data analysis, an email questionnaire was sent to participants asking additional questions about specific topics of interest. The questionnaire contained five questions (see Appendix A) about the role of model-based approaches in AC confidence, the role of formal methods or logic in AC confidence, and the scope of the AC(s) they had prepared. Participants were asked to provide a written response to the questions via email. A small honorarium

was offered as an incentive to reply for this second phase. As with the interview transcripts, written responses were de-identified and subjected to the same open-coding protocol as for the interview transcripts.

3.2.3 Data Analysis

Data analysis involved several steps: 1) cleaning, 2) open coding, and 3) post-coding analysis. These are discussed in turn below.

Transcript Cleaning

After de-identification of the transcripts and email responses, two cleaning tasks were performed. First, filler or pause words (e.g., "umm", "hmm") were removed when they were the only text on a given line. Second, for interview transcripts, for spoken responses longer than a few seconds, Zoom split the text across multiple lines, even if the responses were part of the same topic. The transcripts were manually reviewed and multi-line discussions were combined into one line.

Open-Coding

Open coding was used to analyse the interview transcripts and email responses. Two analysts independently reviewed each line in each cleaned transcript/email and assigned between one and three codes from a shared code book. All coding was performed by the same two analysts. Code assignments did not have order or precedence. Analysts only coded responses from the interviewees, but they could read the full transcript for context. Analysts were permitted to add new codes to the code book as new concepts were encountered or to extend an existing definition. The code book had two levels of hierarchy: a "category" and then "detailed codes". The code book was initially seeded with a handful of codes based on my own experience. Spreadsheet software was used for open coding.

After independently reviewing each transcript and email response, the two analysts met to discuss the assigned codes with a focus on differing code assignments. After discussion, analysts could either change their assigned codes or leave them as they were (i.e., "agree to disagree"). In many cases, the discussion between analysts was productive, leading to deeper insights.

Interview questions #10 and #12 in the interview were multiple choice with a discrete set of answers. These questions were included to both gather concrete data

and stimulate additional discussion. The responses were still coded by both analysts (as above), but a pre-defined set of codes were used. In most cases participants provided both a short answer (their choice) and then a longer form justification, which was also coded.

Agreement Among Analysts. Agreement between the analysts was calculated two times: (1) after independent analysis, and (2) after discussion. The agreement needed to account for the possibility that the raters assigned different numbers of codes to a line in the transcript. Two measures of agreement were used: (1) percent agreement, and (2) Krippendorff's Alpha, with MASI distance for the difference function [169, 170]. The percent agreement calculation was as follows: for each line in each transcript, compute the proportion of agreed codes to the total number of unique codes assigned for the line; then, calculate the average agreement over all lines in all transcripts. For example, for a single line, if analyst R1 assigned codes A, B, C and R2 assigned codes B and D, then the agreement would be $1/4 = 0.25$ since there is one agreed code and four unique codes on the line. The percent agreement calculation, while simple in its calculation and interpretation, does not account for the possibility that raters agreed by chance. So, Krippendorff's Alpha was also calculated.

Post-Coding Analysis

Post-coding analysis proceeded in three steps, and was carried out using both spreadsheets and Python scripts using Jupyter Notebook. First, the coded transcripts and email responses were filtered to remove lines that both analysts agreed were unclear or contained unrelated discussion. Second, conceptually similar codes in the code book were combined. Third, each participant's data was collapsed into a "bag of codes" that described the participant's experiences and opinions. For a detailed code to be included in the participant's "bag", it must be assigned to at least one line in the transcript or email responses, with both analysts agreeing. Themes from all N bags were analyzed within each top-level code book category. Quotations from the transcripts were also extracted to enrich results.

Code Saturation. Saturation is one measure that can show that a grounded theory study is complete for its purpose. Code saturation was calculated to determine whether the interviews and subsequent analysis had reached a point where performing additional interviews would be unlikely to result in novel data being encountered [171, 172]. For the present study, the level of code saturation was calculated by

counting the number of novel codes that appeared in the (agreed) set of codes for each participant, compared to the codes that were used previously. As recommended by [173], near-saturation is achieved when 90% of all codes were used and 100% saturation is achieved when all codes were used.

3.3 Study Results

This section presents the results from recruitment, participant demographics, and the open coding analysis.

3.3.1 Recruitment and Participant Demographics

During recruitment 29 invitation emails were sent, from which 25 people responded indicating interest in participating (response rate of 86%). Four of the respondents did not satisfy the inclusion criteria. The remaining 21 respondents were invited to schedule an interview, from which 19 interviews were conducted. One respondent did not reply to our request to schedule an interview and the other was unable to attend the agreed-upon interview time. During the interview, all $n = 19$ participants were (re-)confirmed to satisfy the inclusion criteria for the study.

During the second phase of data collection, all 19 participants were invited to respond to additional questions via email. Ten participants responded to our questions (response rate of 53%). Two participants replied to our request and explicitly declined to participate in this second phase of data collection due to time constraints. The remaining seven participants did not reply to our email.

Participant Experience. During interviews, participants described their number of years of professional experience, both for general activities (not necessarily assurance related work) and for systems assurance. Overall, the participants had an average of 23 years of general experience and 16 years of systems assurance experience. Additionally, seven participants had at least 35 years of general experience.

Participant Location. Though geography was not a key factor considered while sampling for this study, participants represented a range of locations: North America ($n = 13$), the United Kingdom ($n = 4$), and Europe ($n = 2$).

Industry Representation. Participants were asked to describe the industries they have worked in, both for general activities (not necessarily assurance related) and

those where they have contributed to ACs. The automotive industry (including autonomous vehicles) had the most representation ($n = 9$). Other represented areas were aviation ($n = 7$), defence ($n = 6$), health and medical devices ($n = 4$), marine ($n = 2$), mining / oil & gas ($n = 4$), nuclear ($n = 3$), rail ($n = 6$), robotics and automation ($n = 3$). Many participants reported experience in multiple industries.

Participant Roles. Preparing an AC involves contributions from multiple people with different roles. Participants were asked what role(s) they had assumed while contributing to ACs. The most common role was "reviewer" (i.e., peer or internal reviewer), with most ($n = 16$) participants indicating they had experience reviewing ACs. Further, a majority of participants ($n = 14$) also had experience contributing as a "developer" (i.e., author, creator). Less common roles included "approver" ($n = 5$), "manager" ($n = 4$), and "auditor" ($n = 8$). Many participants reported experience with multiple roles.

3.3.2 Open Coding Results

After cleaning the interview transcripts and email responses, and filtering out questions from the interviewer, there were 758 lines[1] (685 from interviews, 73 from email responses) of content from participants for open coding.

On average, each interview transcript had 36 lines with an average of 75.7 words per line. During coding 100 lines were excluded from the analysis: 95 lines contained content that both analysts agreed was unrelated (e.g., off-topic discussion, greetings, etc.); and 5 lines contained unclear discussion that the reviewers could not interpret clearly enough to assign codes. The final data set for analysis consisted of $n = 658$ transcript lines.

An overview of the final code book is shown in Table 3.1. Codes were organized into a two-level hierarchy with 67 detailed codes distributed over seven categories. In total there were 707 code assignments over the seven categories. Two additional categories (*Impact* and *Importance*) were used to code the responses of multiple-choice questions. The most common category of codes used was *Methods* (assigned 251 times) followed by *Barriers* (assigned 135 times).

[1]The term "line" is used to refer to a sequence of related words, phrases, or sentences; each transcript line or email response might take up many lines on a printed sheet of paper.

Category	Codes	
	# Detailed Codes	# Uses
Motivation for Preparing ACs	10	69
Quality Attributes of ACs	4	59
Scope of ACs†	4†	10†
Expression of ACs	7	60
Methods for Assessment	14	251
Uses of Assessment Results	7	75
Barriers to Conducting Assessments	14	135
Impact of Confidence Assessment*	3*	28*
Importance of Confidence Assessment*	4*	20*
Total	67 (61)	707 (659)

Table 3.1: Overview of final code book (* denotes data collected from a multiple-choice question, † denotes data only from email questionnaire).

Agreement and Saturation

Towards the validity of coding procedure, several agreement and saturation measures were calculated.

Percent Agreement. Percent agreement between analysts was computed before and after meeting for discussion at both the categorical and detailed code level. Before discussion, the percent agreement was 73% per line at the categorical level and 58% per line at the detailed code level. After discussion, agreement was 98% per line at the categorical level and an average of 97% per line at the detailed code level. For reference, for the final code book consisting of 67 detailed codes, if codes were randomly assigned with equal probability, the agreement per transcript line would be 1.70%[2].

Krippendorff's Alpha. Krippendorff's Alpha, using MASI distance for the difference function, was also calculated. Before discussion, agreement was $\alpha_{detailed} = 0.50$ at the detailed code level and $\alpha_{cat} = 0.64$ at the categorical level. After discussion, agreement rose to $\alpha'_{detailed} = 0.96$ at the detailed code level and $\alpha'_{cat} = 0.97$ at the categorical level.

[2]We verified this using a statistical simulation of our agreement calculation over a 100,000 transcript lines that had randomly assigned codes, including accounting for the possibility that analysts picked different numbers of codes.

Saturation. Code saturation was computed. After the third interview, 60% of codes had been used. Near-saturation (90% of codes used) was reached after the 11^{th} interview. Complete saturation (100% of codes used) was reached after the 18th interview. This level of saturation suggests that performing additional interviews (with the specified protocol) would be unlikely to produce novel information about AC practitioners' use of CAMs for the purpose of theory building.

3.3.3 Coded Results

This sub-section discusses the coded results for each top-level category in the code book. Quotes from participants are provided to enrich the results. In cases where two or fewer participants discussed a topic ≤ 2 is reported to protect their privacy.

Objectives for Preparing ACs

Participants were asked questions about their motivation for preparing ACs, the scope of the ACs, and the quality attributes assured by their ACs. Understanding practitioner objectives for preparing ACs helps to contextualize results about CAMs presented later.

Motivations. The *Motivation* category identified ten reasons to prepare an AC, with each participant describing an average of 2.58 reasons. A common motivation for preparing an AC was to achieve compliance ($n = 12$) with technical standards, regulatory requirements, or contractual obligations. While compliance is an initial factor motivating ACs, practitioners recognized other benefits. One participant said: *"Initially, it [the motivation] was purely that the standard requires it. ... Since then, of course, I have realized that [an AC] is much more important than that."* Other common motivations included: capturing arguments in a structured manner ($n = 12$); maintaining a *"live" model of assurance* ($n = 6$) that can be updated as the system evolves; assessing evidence adequacy ($n = 4$); achieving a holistic view of assurance ($n = 4$); a lack of prescriptive guidance for assuring novel technologies ($n = 3$); risk management ($n = 3$); performing gap analyses ($n \leq 2$); managing assurance for complex systems ($n \leq 2$); and supporting deployment decisions ($n \leq 2$).

Quality Attributes. Participants were asked to describe the *quality attribute(s)* that were the focus of the ACs they have worked with. A strong majority ($n = 16$) of participants indicated they had prepared an AC where *safety* was the main quality

attribute being considered. Only four participants ($n = 4$) had prepared an AC where *security* was the primary quality attribute. However, many participants ($n = 11$) had experience preparing ACs that mixed many quality attributes, including safety, security, reliability, availability, fitness-for-purpose, and compliance. Finally, two participants ($n = 2$) indicated they had worked with at least one AC that mixed safety and security, with one describing the difficulty of this approach.

Scope. In the email questionnaire (10 responses total), participants were asked to characterize the scope of the ACs they had prepared. Nine participants had prepared ACs at the "whole system" level ($n = 9$ out of 10); two or fewer participants had prepared ACs for a "sub-system" ($n \leq 2$ out of 10); two or fewer participants had prepared an AC for a "component" ($n \leq 2$ out of 10); and two or fewer participants had prepared ACs for software-only elements ($n \leq 2$ out of 10). Four participants reported having prepared ACs for multiple scopes.

Expressing ACs

The *Expression* category considered notations or methods used by practitioners to represent and document ACs. One can broadly distinguish between structured notations (e.g., GSN) and narratives (e.g., written safety reports). Many participants described experience with both structured notations and narratives ($n = 11$), though not necessarily combining them. All participants described at least one method for expressing ACs, with an average of 2.26 per participant. The distribution is shown in Figure 3.1.

A strong majority ($n = 17$) reported experience with structured notations (GSN: $n = 15$, CAE: $n = 6$, EA: $n = 5$, SACM: $n \leq 2$). A minority of participants ($n = 6$) said structured notations were the only method used to express ACs. Many participants had positive views on structured notations, indicating that they help to organize assurance information. However, some ($n = 4$) participants expressed negative opinions of structured notations. For instance, one participant raised concerns about syntax and semantic details distracting from the essence of assurance arguments. Narratives have been used by a majority of participants ($n = 13$) with a small number ($n = 2$) having used exclusively narrative expression.

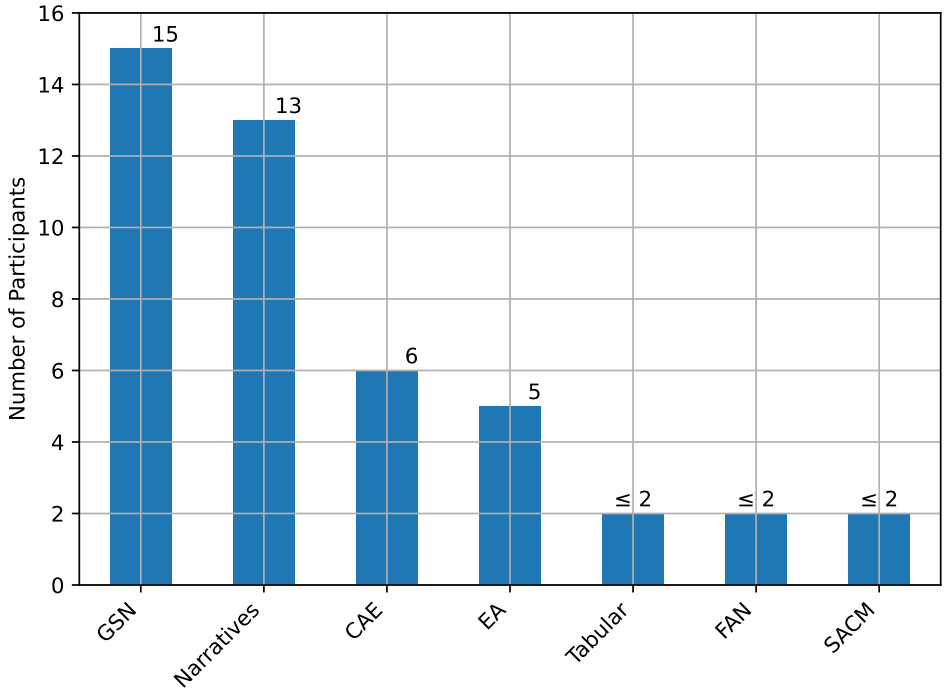


Figure 3.1: Study participants who have used each method of expressing ACs.

Methods for Assessing Confidence in ACs

This study adopted a broad definition of "method" such that the *Methods* category considered methods, techniques, activities, practices, or procedures that have been used by practitioners to assess confidence in an AC. For example, conducting a review activity of an AC was coded in this category, even if a specific review methodology with prescribed steps was not described by the participant. Figure 3.2 shows a breakdown of the methods used by participants. All participants described using at least one CAM, with an average of 3.73 per participant.

Method: Review. Every participant ($n = 19$) indicated they had used some form of peer review. Many described review, and the resulting dialogue, as an essential activity. However, some participants also described concerns with review. For instance, a participant noted: *"... we put a lot of store in peer review. Probably more store than I'm comfortable with ..."*. Some participants ($n = 5$) described a systematic activity where the reviewer examines each reasoning "step" in the argument to determine if the premises (child claims) justify the conclusion (parent claim), which was coded as *interrogation*. When participants described interrogation, it was usually contrasted

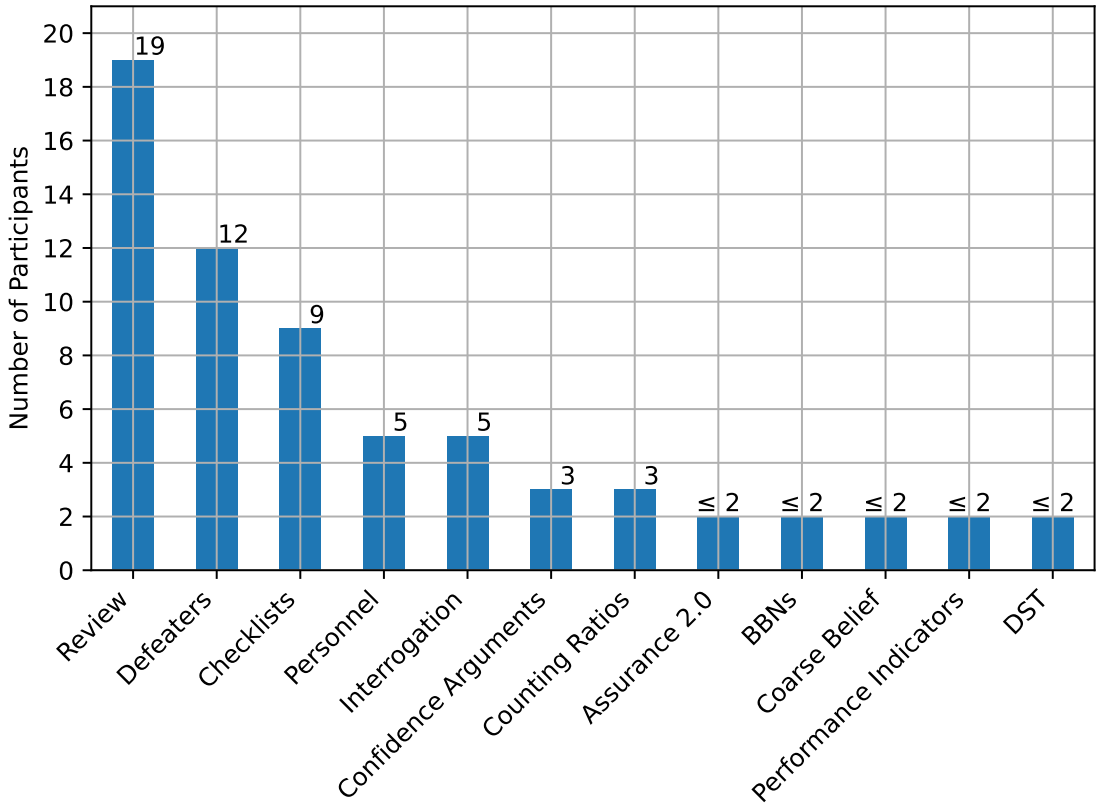


Figure 3.2: Study participants who have used each CAM.

with editorial reviews such as checking for typos in the argument.

Method: Defeaters. A majority of participants ($n = 12$) had used dialectic arguments to challenge arguments or evidence. These were collectively coded as "defeaters". There are several reports of dialectic argumentation for real-world systems [115, 53, 174, 118], so it is not surprising that this is represented in the interview results. One participant described the impact that using defeaters had on their organization's ACs: *"...for the first time, we used 'eliminative induction', 'doubting', or whatever. And it was remarkable. We discovered something like 25 problems that we had not seen previously."* While participants generally described their experience with defeaters as valuable, they identified at least three challenges with their use:

- **Stopping Rules** - Since defeater identification is an open-ended activity, it is not always clear when to stop identifying new defeaters, i.e., there are no clear "stopping rules".

- **Handling Resolved Defeaters** - Once resolved, it is not clear whether defeaters should be left in the argument or removed. On one hand, defeaters capture problems and their resolution, which some participants viewed as increasing confidence in the argument. But other participants noted that leaving defeaters in the argument (even if they are marked as resolved) invites undue criticism by those not familiar with dialectic techniques.
- **Claim v. Defeater** - It can be difficult to determine whether a concept should be framed as a defeater or as a positive claim. It is often possible to adjust the wording of a defeater to make it a claim, and vice versa.

Method: Checklists. About half of the participants ($n = 9$) indicated they had used "checklists", generated from guidance documents or standards, as a means of assessing confidence in the case. In many cases, internationally recognized standards (e.g., UL 4600 for autonomous vehicles [34]) were used as the basis for the checklist, and in other cases internal organization-specific checklists were used.

Method: Quantitative Assessments Participants were asked about their knowledge of, experience applying, and opinion of quantitative CAMs. Table 3.2 shows their reported level of awareness and opinion of quantitative CAMs. Most participants ($n = 13$) indicated they were aware of quantitative CAMs but had not applied them to real-world systems. In fact, only two participants ($n \leq 2$) reported using a quantitative method for a real-world system and they also reported using qualitative CAMs. This suggests that a mixture of quantitative and qualitative CAMs might be necessary. A majority ($n = 13$) of participants expressed a negative opinion of quantitative CAMs. Reasons for negative opinions included:

- **Trust in Methods** - Participants expressed a lack of confidence that the CAMs produce trustworthy results. For instance, one participant said *"a number with no basis in reality is worse than no number."* That is, since quantitative methods are trusted in many other areas of engineering and science, non-expert interest holders might mistakenly trust an unreliable number without considering the argument itself.
- **Lost Information** - Concerns that ACs contain more nuanced information, which cannot be easily distilled down into a handful of numbers.

		Opinion of Quantitative Methods				
		Negative	Positive	None	Unclear	**Total**
Awareness of Quantitative Methods	Not Aware	1	0	2*	0	3
	Aware, Not Used	11	0	0	2	13
	Aware, Used	0	1	0	1	2
	Unclear	1	0	0	0	1
	Total	13	1	2	3	19

Table 3.2: Awareness v. opinion of quantitative CAMs among study participants (* denotes "no opinion because no awareness").

- **Interpreting Results** - It might be difficult to justify the resulting confidence value to an independent assessor or regulator, especially if they are not familiar with quantitative CAMs. One participant said *"... if we ever had to go and argue this in front of [a regulator], ... with 86% confidence that the system wouldn't [harm someone] ... it may become more difficult to defend that kind of thing ..."*

- **Selecting Inputs** - There are challenges identifying the inputs to quantitative CAMs. Distilling the information in the argument's leaves into numbers might be error-prone, especially if users do not understand what each input means.

Only one participant reported a positive opinion of quantitative confidence assessment: *"The Bayesian [method] it seems, is much easier for experts to give you an assessment of a post-condition on something."* In other words, to determine the final assessment (the "post-condition") asking analysts to react to a potentially inaccurate assessment of confidence (a "prior") produces more useful assessments.

Method: Modelling. The email questionnaire asked participants about the role that model-based engineering approaches (such as SysML) have had on their confidence in an AC. Of the ten email responses, six ($n = 6$ out of 10) said they used models in combination with an AC, four respondents had not used models in combination with an AC. Four respondents indicated that their use of models had a positive impact on AC confidence and two indicated the impact had been neutral. Notably, respondents distinguished between using models as evidence and using it to inform the AC's argument structure. Only one respondent described using models to assess the argument structure itself. It was also observed that using models to gain

confidence in an AC does not necessarily increase overall confidence in a system, but rather "shifts" the burden of confidence to another aspect of the project (i.e., the model).

Method: Formalization. The email questionnaire asked participants about the role that formal methods (e.g., satisfiability solvers or theorem provers) have had in their confidence in an AC. Of the ten email responses, only two or fewer ($n \leq 2$) respondents had used formal methods to assess ACs. Further, respondents expressed mixed views about the potential utility of formal methods for ACs: four ($n = 4$ out of 10) thought that they might be useful, two or fewer expressed a negative opinion, three expressed uncertain or mixed opinions, and two did not respond clearly to the question. Additionally, during the interviews, two participants expressed negative opinions of using formal methods for ACs.

Uses of Confidence Assessment. Participants reported seven *Uses* of the results produced by CAMs. All participants reported at least one use, with an average of 2.84 per participant. A majority ($n = 14$) indicated that they have used CAM results to communicate with interest holders. For instance, the results of a review or checklist activity might be shown to an organization's decision makers as a way of communicating the maturity of the AC. This is related to the goal of understanding (qualitative) risk associated with the system ($n = 8$). Some indicated they had used CAM results to evaluate the argument ($n = 7$) and supporting evidence ($n = 6$) and to address issues ($n = 8$) to improve the AC's quality. Avoiding confirmation bias was also identified ($n = 5$). Finally, some participants ($n = 6$) used CAM results to support deployment decisions.

Barriers to Confidence Assessment

Participants were asked to discuss reasons that a CAM might not be used (i.e., "barriers"). In total, 14 *Barriers* were identified, several of which are discussed below. All participants identified at least one barrier, with an average of 4.47 per participant. Figure 3.3 shows the number of participants who identified each barrier.

Barrier: More Work. A majority ($n = 13$) of participants identified that "more work" (i.e., additional effort) is a barrier to using a CAM. In reference to the argument interrogation method, a participant said that *"...if you are writing a healthy GSN safety case and you ask people to look at every decomposition, that can start to get*

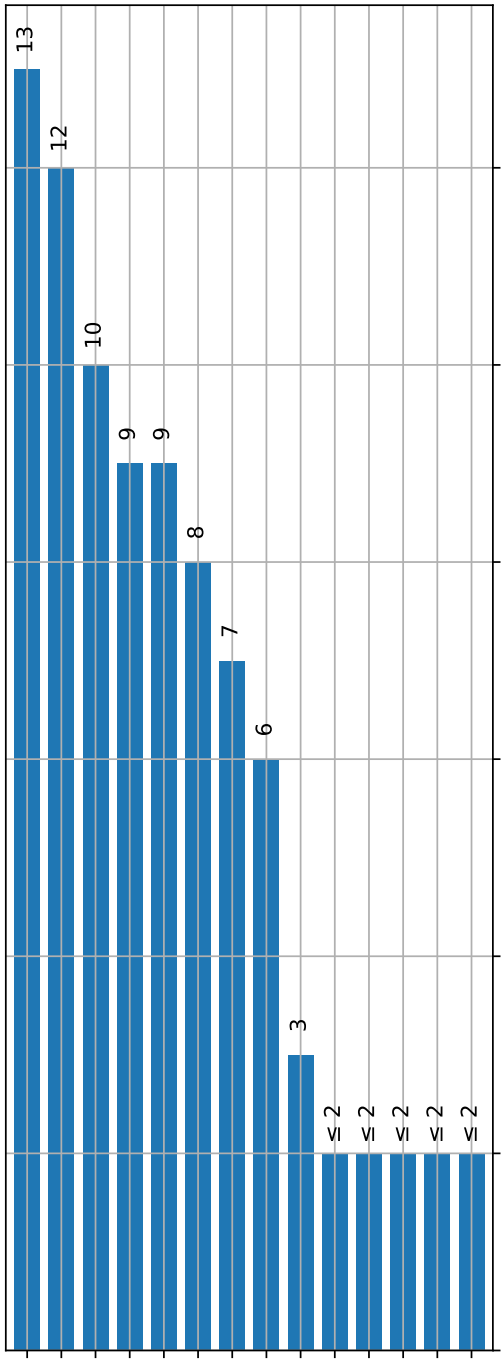


Figure 3.3: Number of participants who identified each *barrier* to performing AC confidence assessment(s).

quite labour intensive." Several participants observed that it might be difficult to justify the extra effort relative to schedules and budgets. Finally, some participants noted that quantitative CAMs require many numerical inputs: one said *"[The] level of effort to use methods could be high. Especially if [you are] needing to put many numbers all around."* Indeed, some quantitative methods require between two and four inputs per node in the AC [50, 48]. A real-world AC such as the one in [174], which contains on the order of 500 nodes, would require significant effort to select these inputs.

Barrier: Inadequate Guidance. Many participants ($n = 12$) felt that CAMs are inaccessible to practitioners due to inadequate published guidance or training materials about their application. Concerns included:

- **Methods are Hard to Understand** - To use a CAM, practitioners need to understand the underlying theory, but the description of some CAMs in the literature can be hard to understand. This is especially true for quantitative CAMs that are based on unfamiliar mathematical frameworks. This problem was clearly articulated by one participant: *"... you don't necessarily understand how the confidence algorithm is working"* and again by another: *"... people need to get up a really big learning curve before you can actually apply [a quantitative CAM]."*

- **Difficulty Selecting a Method** - Since there are multiple CAMs available, it can be difficult to know which method to use. A participant described this well: *"I would like to know which one to use when. ... there might be some that are really good for one thing and not another. I don't know which one."*

- **Missing from Engineering Curriculum** - One participant noted that ACs and their assessment are not typically covered as part of the standard system or software engineering courses offered by educational institutes. This suggests there will continue to be a knowledge gap in this area for new graduates.

- **Lack of Authoritative Guidance** - Technical standards and regulations do not provide enough guidance on how to construct them or assess confidence: *"I want to blame a little bit [technical] standards, because standards they actually say to [prepare an AC], and they didn't tell how you can [assess confidence]."* Since compliance is necessary for practitioners, providing clearer guidance for performing assessments would help practitioners.

Barrier: Interpreting Results and Subjectivity. Just over half of the participants ($n = 10$) were concerned with interpreting results from quantitative CAMs. For example, how should a confidence score of 0.82 be interpreted? Is it enough to deploy a system? What if, after more effort, the score increased to 0.87, is that sufficient? Several participants identified a related concern about the subjectivity of both quantitative and qualitative CAMs ($n = 9$), that might cause practitioners using the same CAM to produce different results. Indeed, practitioners all have unique experiences and knowledge that might cause them to weight arguments or evidence differently. For quantitative CAMs, one participant expressed this as: *"... I put some numbers, and you put totally different numbers, and then who do we trust?"*

Barriers: Confidence in Confidence and Immature Methods. Several participants ($n = 9$) described a meta problem with CAMs: how can they be confident that the CAM produced a trustworthy result (i.e., "confidence in confidence")? For quantitative CAMs, concerns were focused on interpreting results, subjectivity (see above), or their maturity. Several participants ($n = 8$) expressed concerns about the maturity of quantitative CAMs. For qualitative CAMs, participants also raised concerns about completeness. For instance, if dialectic argumentation is used, it is not clear when to stop identifying defeaters: *"[I have] started to [use defeaters], and I do find it useful. I think that there are challenges with recording it and with the stopping rule ... "*.

Barrier: AC Adoption and Safety Culture. Some participants ($n = 6$) suggested that the adoption of ACs remains a challenge. That is, it is not possible to apply CAMs without first preparing an AC to assess. Further, some participants ($n = 7$) identified issues related to safety culture as a barrier to applying CAMs: the results of a confidence assessment might be unsavory, and identify problems that require more effort to address. For instance, a participant said *"... the safety case or safety analysis or even the safety culture is seen as a hindrance rather than a benefit."*

Impact and Importance of Confidence Assessment

Participants were asked two multiple choice questions to capture concrete opinions about the impact of CAMs and their perceived importance. In addition to capturing direct answers, these questions served to elicit further discussion.

Impact. The first question was about the impact that CAMs had on real projects from the participants' experience. Overall, CAMs had a positive ($n = 17$) impact on projects. Only one participant ($n = 1$) said it had a neutral impact. Three ($n = 3$) participants had experiences where CAMs had a negative impact. These negative experiences were limited to specific projects, and the same participants also described other projects where CAMs had a positive impact. One participant did not clearly respond to this question.

Importance. The second question asked the participants about the importance of CAMs and the extent to which they should be prescribed in the future. Almost all participants felt that confidence assessment should be either *mandatory* ($n = 8$, 42%) or at least *recommended* ($n = 8$, 42%). Only one participant ($n = 1$, 5%) thought it should be *optional* and none thought it should be *discouraged*. One participant did not clearly answer this question. Many participants who answered *recommended* or *optional* indicated that they would have picked *mandatory* but for the fact that there are no minimum criteria or widely recognized CAMs (except for "review") and so it is unreasonable to require them without widely recognized methods.

3.4 Related Empirical Studies

As noted in the introduction, there are relatively few empirical studies on ACs. With some exceptions (e.g., [91]), it generally is difficult for researchers to study full ACs. Therefore, most empirical work (including this chapter) focuses on collecting data from practitioners. While other researchers have previously engaged practitioners, to my knowledge, this is the first empirical work on the use of CAMs by practitioners. This section reviews other studies that collected data from AC practitioners and compares the results with those from the interview study. A summary is provided in Table 3.3.

Törner and Öhman 2008. This study used interviews to survey ten engineers from three Swedish automotive companies about the adoption, use cases, and issues for safety cases. They concluded that there is a need for ACs for automotive technology [175]. Their survey was conducted before 2011, when the first edition of ISO 26262 was published, which required the preparation of an AC for electronics and software in automotive systems. Therefore, the survey's results provide some indication of the automotive industry's perspective on ACs prior to their inclusion in a

Study	Year	Method	n	Focus	Relevant Result(s)
Törner and Öhman [175]	2008	Interviews	10	Adoption and uses of safety cases by Swedish automotive manufacturers.	ACs are useful as communication aids and for documenting arguments, but practitioners were concerned about the added work required to prepare ACs.
Nair et al. [78]	2012	Questionnaire	52	Evidence management for critical systems and comparison between practices in industry v. academic literature.	Textual methods are used for expressing ACs, but there is interest in structured notations (e.g., GSN). Assessing confidence is important, but remains challenging for practitioners.
Doss and Kelly [176]	2016	Questionnaire	31	Relationship between safety engineering activities (incl. ACs) and agile process approaches.	Iteratively developing and (re-)evaluating an AC is an important task for safety-critical systems.
Cheng et al. [76]	2018	Interviews	9	Understanding practitioner perception(s) of ACs.	ACs are useful as a communication aid. There are challenges related to scalability, practitioner expertise, confidence management, and tool support.
Almendra et al. [177]	2022	Questionnaire	31	Interplay between requirements and ACs.	Creation of ACs is driven by regulatory or organizational requirements. Structured methods for expressing ACs (e.g., GSN) are used by many practitioners.
(this study)	2024	Interviews	19	Methods and barriers for AC confidence assessments performed by practitioners.	N/A - *This is the present study.*

Table 3.3: Summary of previous empirical studies on AC usage by practitioners.

landmark technical standard. Key use cases identified by Törner and Öhman included communication, documentation, and aiding early system development. Their participants also worried that preparing an AC would result in an "increase in workload" [175]. Even after 16+ years, some of Törner and Öhman's findings are consistent with our work. First, using an AC as a communication tool remains an important theme. Second, we also found that documenting assuring arguments is a motivation for preparing ACs. Finally, the concerns about increased workload associated with preparing an AC (or performing confidence assessment) is a shared result.

Nair et al. 2015. This study used a web-based questionnaire to survey 52 practitioners about evidence management and compared the results with trends from the literature [78]. Their results show that, at the time, structured notations (e.g., GSN, CAE, etc.) for expressing ACs and organizing evidence were rarely used by practitioners who preferred textual methods (e.g., narratives). However, there was

interest from practitioners in structured notations as a means of organizing assurance evidence. Nair et al. also reported that using arguments to connect evidence to compliance requirements from technical standards was important. Twelve years later, the results from our study suggest there is increased adoption of structured notations. Regarding confidence assessment, Nair et al. also found that evaluating how well evidence supports a claim was an important challenge, which is consistent with our results about the importance of confidence assessment.

Doss and Kelly 2016. This study used a questionnaire to survey 31 practitioners to investigate challenges and opportunities for applying agile methods to safety-critical systems; part of their survey focused on ACs, and they found that 90% of participants recognized the importance of regularly (re-)evaluating an AC [176]. Based on their results, Doss and Kelly proposed that incrementally developing an AC in an agile approach would be beneficial. Some participants in our study identified that having a "live" model of assurance was a motivation for preparing ACs. This is lower than reported by Doss and Kelly. However, the purpose of their survey was to investigate the use of agile methods, whereas our study did not provide the same conceptual framing, so it is not surprising that Doss and Kelly found this to be a stronger theme. Nonetheless, it is still a shared theme between the two works.

Cheng et al. 2018. This study interviewed nine experts with experience developing safety-critical systems and preparing ACs [76]. They identified several benefits of ACs related to communication, organization, and assuring novel technologies. Their finding that ACs are used to support communication is consistent with our results. Cheng et al. identified several challenges associated with ACs, including scalability, change management, practitioner expertise and skills, dealing with complex systems, managing uncertainty and confidence, the method being "too flexible", and handling incomplete information. While the focus of Cheng et al.'s study differed from ours, there are similarities in the challenges reported by practitioners. First, Cheng et al. identified managing uncertainty and confidence as a challenge for practitioners, which is the focus of our present study. Our results suggest that this is far from a settled question from the perspective of practitioners. Second, the challenge of scalability appears in our results in terms of the number of inputs required for quantitative CAMs. Third, their findings about practitioner expertise and skills parallels our own results about inadequate guidance being available to practitioners.

Almendra et al. 2022. This study used a web-based questionnaire to survey a population of 31 practitioners and researchers about the interplay between requirements engineering and ACs [177]. As part of their questionnaire they asked questions about the motivations for, and uses of, ACs. Their results show that AC development is thought to be beneficial, but that it is mostly driven by regulatory requirements. This is consistent with our own finding about compliance being a leading motivation for preparing ACs. In terms of expressing arguments, Almendra et al. reported that a slight majority of their participants used GSN to structure ACs (52%), with a smaller proportion (26%) using textual narratives. Our results are partially consistent with theirs in that a majority of our participants used structured notations, but that a notable number also used narrative expression.

3.5 Limitations and Threats to Validity

As with any research, this study is subject to several limitations and threats to validity that should be considered when interpreting the results.

3.5.1 Theory Building v. Theory Validation

This study used a grounded theory method: through interviews, an email questionnaire, and open coding analysis a detailed understanding of how the study participants think about ACs and confidence was developed. The result is a "theory" (grounded in data) about current practice. While diversity in the participants helps improve the likelihood that this theory will generalize to a wider population of AC practitioners, this has not been empirically confirmed. Future studies should test hypotheses that are predicted by our present results.

3.5.2 Sampling Method and Sample Size

This study used a modest sized "convenience sample" from the authors' professional networks. A more robust sampling strategy would offer more robust empirical results. However, several factors are worth considering in this regard.

- Recruitment sought participants with a range of industry experiences, and also included second (and even third) degree connections from the investigators networks.

- The interviews reached 90% of code saturation after eleven interviews and 100% code saturation after 18 interviews. That is, after the eleventh interview, most of the information on this topic had been uncovered and the subsequent seven interviews had diminishing returns in terms of theory building. Additional interviews using the same instrument and analysis protocol (beyond the 19 of this study) would be increasingly unlikely to yield new information.

- From Table 3.3 it can be seen that the current study is the largest set of interviews among previous empirical works on ACs. Previous works by Törner and Öhman and Cheng et al. had 10 and 9 participants respectively.

- The population of AC practitioners is relatively small compared to other practitioner populations that are often studied in systems and software engineering. For instance, the population of software developers (or even sub-sets like embedded software developers) is significantly larger than that of AC practitioners. This is likely a result of the ACs being a rather niche topic, requiring the intersection of several factors, including knowledge specialization in safety engineering and working in a high-risk application area (meriting an AC).

Despite the above considerations, the sampling method precludes statistical inferences about whether the results represent the whole population of AC practitioners. Robust sampling methods, which are required for generalization, are impractical to apply for this population since there is no way to enumerate a complete population from which to sample [178]. Future empirical studies that seek to validate our findings would need to take this into account.

3.5.3 Limitations of Interview Methods

While structured interviews provide an opportunity to collect rich data, they also limit the completeness of the results. During interviews, it is possible that practitioners described their use of CAMs based on what was "on their mind" without considering a comprehensive list of potential answers. The interview questions could have been more specific (e.g., "have you used method X"), but this could have also impacted the results by leading participants towards a pre-determined result. Since our objective was one of theory building, not theory validation, we erred on the side of open-ended questions. It is possible that using more multiple choice questions might have produced different results.

3.5.4 Other Approaches to Risk Management

The target population for this study was AC practitioners, i.e., individuals that prepare ACs for real-world systems. As a result, the inclusion criteria for the study required that participants have experience preparing an AC. For instance, at least two of the candidates who responded to the study's invitation did not meet this criterion, though they did have experience working on assurance for high-risk systems. It is likely that there is a population of practitioners who do not work with ACs and use different methods for gaining confidence in their work products, which were not captured by this study.

3.6 Synthesizing the Practitioner Perspective

This study provided insight into the current state of practice for ACs and confidence assessment across several topic areas, represented as categories in the code book. Towards the objective described at the beginning of this chapter, these results are synthesized to create the "Practitioner Perspective" on confidence assessment.

3.6.1 Objectives for Preparing ACs

Compliance with regulations, technical standards, or contractual requirements is a principal motivator for preparing ACs. Practitioners also recognize that ACs are useful for documenting assurance arguments in a structured manner, for communicating with interest holders and decision makers, and evaluating evidence. Some view ACs as "live" models of assurance that evolve alongside a system. Most ACs are prepared with safety as the primary quality attribute. However, some ACs also consider security or a combination of several quality attributes (safety, security, reliability, fitness-for-purpose, etc.). The scope of ACs is varied and can cover whole systems to individual components or even elements that make up components (e.g., a software library).

3.6.2 Expression of ACs

Structured notations (e.g., GSN, CAE, EA, SACM) are now widely used to express assurance arguments. Further, despite wide use of structured notations, narrative expression of ACs is still common, with many practitioners having experience with

both structured notations and narratives. However, some concerns exist related to the complexity of structured notations distracting from the essential purpose of ACs as a means to clearly capture evidence-based assurance arguments. That is, there is a concern that users of structured notations become overly focused on producing a "notationally correct" version of an argument at the expense of clarity, which undermines the value of ACs as a communication tool. While understandable, this concern is surprising given that one of the intended objectives of structured notations is to help authors and readers focus on argument structure, without having to wade through lengthy prose. Perhaps the problem is that articulating a clear argument is a matter of craft that cannot be reduced to a notational exercise, a point that is also made by Holloway [32].

3.6.3 Methods for Assessing Confidence

Results from applying CAMs are used to communicate to interest holders, understand the risk associated with the system, and improve the quality of the argument or evidence. The following describes the state of practice for several CAMs, as expressed by the study participants.

Review

Review is a very common[3] CAM, with some practitioners also using systematic review methods to "interrogate" an argument. Checklists derived from standards, guidance documents, organizational practices, and experience are used as review criteria to confirm necessary topics have been addressed. It is not surprising that review is a widely used CAM. Indeed, review is the cornerstone of quality management activities in many fields. However, despite the importance of review, there might be an over-dependence on it as the *only* CAM used, and practitioners might value complementary methods. In light of this finding, additional guidance for practitioners on how to perform meaningful reviews might be valuable.

Dialectic Reasoning

Using "defeaters" is widely viewed as a means of challenging the positive perspective(s) in an AC. This finding is interesting because it has seen faster adoption than

[3]I hesitate to say that review is "universally" used, but it is worth noting that all participants in our study identified review as a method.

other qualitative CAMs (e.g., ACPs) that have existed in the academic literature for just as long, if not longer. One possible explanation is that dialectic methods offer a relatively high reward (e.g., mitigation of confirmation bias) in a manner that can be flexibly incorporated into existing AC practices. Another explanation is that dialectic methods permit AC authors to reason about doubts that they might have otherwise been unable to express as part of formal assurance processes [53]. However, there are open methodological questions about the use of dialectic methods such as the absence of clear "stopping" criteria, handling of resolved defeaters, and distinguishing between genuine argument defeaters versus claims.

Quantitative Methods

While many practitioners are aware of quantitative CAMs, they have seen limited use for real-world ACs. Further, many practitioners have negative opinions of quantitative methods due to: a lack of confidence that the methods produce trustworthy results; loss of information when confidence is expressed as a number; difficulty explaining results to interest holders; and the volume of inputs required. Still quantitative CAMs are an attractive prospect in that they offer a systematic framework to numerically reason about one's confidence. However, the results from this study show that significant work, both in terms of developing new methods and validating existing ones, remains to demonstrate their trustworthiness.

Using Models

Model-based approaches, particularly where a model is the source of evidence, can be used to increase confidence in an AC. The use of models by practitioners to assist with the preparation or management of the AC's argument is less common, and practitioners might hesitate to adopt such approaches, especially if model-based approaches are intended to replace human-led reasoning about the argument. The base adoption for model-based approaches (outside ACs) likely influences their use in combination with ACs. That is, as model-based engineering approaches grow in popularity, then their use will likely also grow for ACs.

Formalizing Arguments

Using formal methods to check properties of assurance arguments (e.g., logical soundness) does not appear to be widely used by practitioners. Some practitioners believe

this could improve the quality of ACs. However, others identified barriers to their use, including challenges translating from natural language to a formal representation that preserves the argument's nuance; concerns about making the argument more difficult to understand for non-experts; and the availability of personnel with the technical skill to prepare and review formal proofs.

3.6.4 Barriers to Confidence Assessment

Several prominent barriers for using CAMs emerged from practitioners.

"Using CAMs is More Work"

CAMs are perceived as adding to the workload of practitioners, especially for quantitative CAMs that require many inputs. While it is true that more effort usually is necessary to apply CAMs, it is worth asking whether that effort adds value to the assurance process or the organization as a whole. Some CAMs have clear value propositions in terms of increasing the quality of the AC, such as review, and are thus widely performed as a necessary activity. To overcome this barrier, advocates for CAMs need to demonstrate that the extra effort adds a proportional amount of value for an organization.

"CAMs are Difficult to Understand"

The existing methods, especially quantitative ones, are difficult to understand. Indeed, some quantitative CAMs use non-trivial mathematical formulations and theories that practitioners might not have encountered in their training. Additionally, this problem is (at least in part) due to a lack of accessible guidance and training material on the theoretical basis for CAMs and how to apply them. Moreover, while it is tempting to focus on creating guidance for AC practitioners, they are only half of the educational challenge: providing guidance for a wide range of interest holders, who are often in a position to interpret results, and have variable expertise, is also essential.

"CAMs are Subjective and Results are Difficult to Interpret"

Confidence assessment is seen as a subjective activity that produces variable results which can be difficult to interpret. This challenge is partially one of education and

guidance (see above). However, it is also worth noting that confidence is an inherently subjective quantity. This is a multi-faceted problem that remains at the heart of research on AC confidence.

"Quantitative CAMs are Not Trustworthy"

Quantitative methods are relatively immature and there is a lack of evidence to show they are trustworthy. In one sense, this barrier is good news for the community of researchers working on AC confidence: there is plenty of work to be done! One way to build trust for these methods is to gather additional empirical data from both real-world and artificial settings that demonstrates they have desirable properties. Graydon and Holloway expressed a similar view [80].

3.7 Chapter Summary

This chapter presented the results from an interview study of 19 AC practitioners representing a range of industries and experience levels. The results of the study were used to develop the "practitioner perspective" on CAMs which will inform work in subsequent chapters in this dissertation.

Overall, practitioners felt that it is important to perform some manner of confidence assessment or confidence building activity when developing an AC. Practitioners were critical of quantitative CAMs, and favoured qualitative CAMs that emphasize (peer or independent) review, dialectic argumentation, and using checklists to guide reviews. Practitioners also identified barriers to performing CAMs, including additional effort, difficulty understanding and interpreting results, subjectivity of methods, and a lack of trust in quantitative methods.

Chapter 4
Requirements for a New Confidence Assessment Method

Towards the goal of developing a new method that enables practitioners to perform confidence assessments on DACs, this chapter combines the practitioner perspective from Chapter 3 with other perspectives to elicit requirements for a new CAM. The *Dynamic Perspective* will consider how a new CAM should be applied to a DAC, and the *Dialectic Perspective* will consider how to integrate dialectic reasoning into the CAM. The requirements identified in this chapter will be used to develop and evaluate a new CAM later in this dissertation. Some parts of this chapter have been previously published in [84], particularly the requirements of the dialectic perspective, extension of an existing CAM, and worked example in Section 4.3.

4.1 Requirements of the Practitioner Perspective

Using the "Practitioner Perspective" developed in Chapter 3, this section defines requirements that a new CAM should satisfy to improve adoption in practice. These requirements are intended to be applicable to both qualitative and quantitative CAMs, though later sections in this chapter focus on quantitative methods.

4.1.1 Support for Structured Argumentation

As reported in Chapter 3, structured notations for expressing AC arguments are widely used. This finding is also consistent with broader trends in the literature, where structured arguments are identified as an essential component for assuring

modern systems, including those that use AI [25, 19]. Given the growth in this area, CAMs should be applicable to ACs expressed using structured notations.

R1 (Structured Argumentation) *The CAM shall be applicable to ACs expressed using structured notations, such as Goal-Structuring Notation (GSN), Claim-Argument-Evidence (CAE) notation, or Eliminative Argumentation (EA).*

It is worth noting that most CAMs described in the literature require the use of structured notation to express arguments, and as a result already satisfy this requirement, e.g., [47, 48, 50]. However, it is included as a requirement because of the prominent use of structured notations by practitioners.

4.1.2 Support for Dialectic Reasoning

As seen in Chapter 3, after peer review, applying dialectic reasoning (i.e., "defeaters") is the most common method of gaining confidence in an AC. Doubting oneself (or peers) has a long tradition in academic and technical fields, and so it is not surprising that dialectic reasoning is prominent among AC practitioners. Moreover, defeaters have also been incorporated into many structured notations, including GSN [20], CAE [46], EA [31], and FAN [32]. CAMs should support concepts that are present in structured notations and that practitioners are naturally using in practice, including support for dialectic reasoning.

R2 (Defeaters) *The CAM shall support users to engage in dialectic reasoning (i.e., using "defeaters", "counter-goals", or similar), including accounting for changes in confidence due to the presence of dialectic elements in the argument.*

Including dialectic reasoning into CAMs is a non-trivial task. Among the challenges is deciding how a defeater changes confidence. This requirement is intentionally worded to permit a wide range of interpretations and will be elaborated upon in Section 4.3 below.

4.1.3 Mitigate Subjectivity

A common concern about CAMs is the subjectivity of the inputs to the method. Multiple experts might weigh evidence or reasoning differently when assessing confidence based on their training and experience. For instance, for software verification one expert might place high value on formal verification activities while another might

be skeptical of their limitations and instead prefer test-based methods. While it is unlikely that subjectivity can be entirely removed from AC assessment, CAMs should include measures to mitigate its impact.

R3 (Subjectivity) *The CAM shall employ measures that mitigate the impact of subjectivity on confidence assessments.*

4.1.4 Interpretable Results

Interpreting the outputs and results of CAMs remains a challenge. While this is especially true for quantitative CAMs, as discussed in Chapter 3, it might also be a challenge for qualitative methods. For instance, when using the ACP method, how does one decide if the dedicated confidence arguments are satisfactory? For quantitative methods, where confidence is expressed numerically, this challenge might manifest when decision makers, who might not be experts in CAMs, are required to interpret a computed confidence level (say 88%) and determine if it is acceptable to deploy a critical system.

R4 (Interpretation) *The CAM shall produce a result that can be readily interpreted by a wide range of non-expert interest holders and decision makers to determine if there is adequate confidence in the AC.*

4.1.5 Understandable Mechanics

Some CAMs depend on formalisms and theories that are difficult to understand for busy practitioners. In order to trust the results produced by CAMs, practitioners must be able to, within limited time, understand the mechanics of the method so that they can justify how a particular result was reached. Ideally, a CAM's mechanics are also understandable (though perhaps with less depth) to non-expert interest holders.

R5 (Understandability) *The mechanisms, calculations, reasoning, or criteria used by the CAM shall be understandable to users such that it is possible to explain how a particular result was produced.*

4.1.6 Expressing Nuanced Reasoning

In Chapter 3, practitioners raised concerns about the loss of nuance when applying quantitative CAMs, i.e., information is lost when translating the qualitative reasoning

expressed in an AC's claims into a numerical representation. Moreover, quantitative CAMs usually have a limited number of configurations for propagating confidence through the argument (e.g., "AND" and "OR"), which results in further losses. CAMs are fundamentally tools to support decision-making, and so they should enable, not limit, users to express their thinking.

R6 (Expressivity) *The CAM shall enable users to capture nuanced or detailed reasoning about their confidence in the AC's argument.*

Notably, the qualitative ACP method naturally satisfies this requirement [47]. When using ACPs, users express detailed "confidence arguments", having an arbitrary level of detail, in support of the main argument.

4.1.7 Method Applicability and Stopping Criteria

In Chapter 3, one of the challenges raised by practitioners is knowing which CAM to use (i.e., applicability criteria) and when to stop applying the method. For instance, in the case of dialectic reasoning, practitioners must decide when to stop searching for defeaters; no explicit "stopping" criteria exist for this purpose. Practitioners' desire for stopping criteria appeared in their use of checklists as a means of checking completeness of ACs, as reported in Section 3.3.3.

R7 (Applicability Criteria) *The CAM shall define applicability criteria indicating prerequisites that must be satisfied prior to application to an AC.*

R8 (Stopping Criteria) *The CAM shall define criteria for when the assessment activity is complete.*

Some methods already provide implicit stopping criteria. For example, the qualitative iTest method has a set of prompts to ask about each step in the argument; once all prompts are considered for each step, it is reasonable to conclude the assessment is complete [49]. As another example, the quantitative BBN method requires the user to assign belief valuations and parameter values to nodes in the argument; once all assignments are provided then the method is completed.

4.1.8 ALARP Effort

The most common concern raised by practitioners was that using a CAM requires "more work". Ultimately, activities intended to gain or assess confidence in an AC will

take *some* effort. The question is whether the value obtained (i.e., higher assurance) justifies the effort. One way to improve this effort-value trade-off is to ensure that the effort to apply a CAM is as low as reasonably practicable ("ALARP"). For instance, the number of inputs required to apply the method, especially in the "average" case[1], should be as low as possible.

R9 (Effort) *The effort to apply the CAM shall be as low as reasonably practicable in the average or most commonly occurring usages of the method.*

For quantitative CAMs, one way to reduce effort in the average use case is to select sensible default input or configuration values, rather than requiring the user to specify each value separately.

4.2 Requirements of the Dynamic Perspective

Based on the Definition 5 for DACs, assessing confidence in DACs is a generalization of assessing confidence in a static AC that considers several additional factors: 1) evolving rationale and arguments, 2) updating evidence, 3) assurance measures with live data, and 4) decision-making based on the revised AC. While these factors are also applicable to a static AC, the dynamic nature of DACs increases the importance of these factors when designing a new CAM for this purpose. Each factor is considered below to elicit requirements.

4.2.1 Evolving Arguments

Throughout the life of a DAC, it is likely that its assurance rationale or argument will be changed. There are many reasons that an argument might change, some include: changes to a system design; changes to regulatory requirements or guidance for assuring critical systems; or changes in the operational environment. Regardless of the reason for change, updates to the assurance argument in a DAC are likely to impact confidence assessments. Moreover, in some contexts these changes might occur (semi-)automatically. For instance, an adaptation made at run-time by a self-adaptive system might require a revision to its DAC's argument.

[1]Not unlike a time-complexity analysis, comparing average v. worst case usage effort to apply a method.

R10 (Evolving Arguments) *The CAM shall provide a partial or fully automated means to revise the confidence assessment in response to changes in the argument.*

When embedded in software tools, some CAMs already offer mechanisms to respond to argument change through APIs. For instance, the BBN method requires several configuration parameters for each claim in the argument; these are easily exposed through an API, and automated changes to the argument structure can update the configuration parameters.

4.2.2 Support for Updated Evidence

As with arguments, the evidence supporting a DAC's argument is likely to be updated. For instance, in a continuous assurance process new test results might be automatically generated when the software for a critical system is revised.

R11 (Updated Evidence) *The CAM shall provide a partial or fully automated means to revise the confidence assessment in response to changes in evidence supporting the AC's argument.*

4.2.3 Indicators

Indicators are computed from "live" data flowing from a variety of sources, including development activities and system operations [64, 92]. Indicators provide on-going validation of claims in the DAC [63]. As with arguments and evidence, indicators are expected to change over time. Ideally, connecting them to a CAM will broaden their impact beyond a single claim or sub-argument, by allowing the effects of a change to propagate through the argument and revising the confidence in the argument's top-level claim.

R12 (Indicators) *The CAM shall provide a partial or fully automated means to revise the confidence assessment in response to changes in indicators connected to the AC's argument.*

4.2.4 Decidability

For each change to an argument, evidence, or indicators, it is necessary to re-assess the confidence in a DAC. To support continuous and perpetual assurance processes, the (re-)assessment procedure should be *decidable* in the sense that, given all necessary

inputs, it should be possible to definitively, ideally automatically, determine the level of confidence in the DAC's top-level claim. That is, the DAC, when instrumented with a CAM, should be viewed as encoding a decision-making procedure about whether a system can be deployed or continue to operate.

R13 (Decidability) *The CAM shall provide a partial or fully automated means to re-assess confidence in the DAC that can be used to support decision-making about whether there is sufficient confidence in the DAC to justify deploying, or continuing to operate, a system.*

Note that both quantitative and qualitative CAMs can satisfy this requirement. In the case of a quantitative CAM, such as the BBN [48] or DST [50] methods, where a series of calculations are performed to compute an overall confidence score, an automated procedure exists that satisfies this requirement. In the case of qualitative methods, additional procedures might need to be developed to reason about confidence in a manner amenable to automation.

4.3 Requirements of the Dialectic Perspective

As noted in the practitioner perspective from Chapter 3 and Requirement R2 above, a new CAM must have built-in support for dialectic reasoning (i.e., "defeaters"). After introducing foundational terminology, this section derives requirements for including defeaters in quantitative CAMs. These requirements are intended for use with quantitative CAMs that accept confidence inputs at the leaves of the argument and propagate confidence upwards through the argument to produce an overall measure of confidence in the top-level claim.

4.3.1 Confidence versus Belief

So far, this dissertation has used the term "confidence" somewhat imprecisely, with only an implicit definition provided by Definition 4. From a quantitative perspective, confidence is a matter of degree and may be represented using a numerical scale, e.g., $[0, 1]$ where 1 represents certainty in the truth of the claim. Having zero confidence in a claim suggests a state of complete uncertainty about whether the claim is true or false. That is, there is no support for or against the claim.

A problem arises with this terminology when considering defeaters in an AC.

Goodenough et al. take the position that, if a defeater raised against a claim is valid, then the claim is thought to be false [31, Table 1]. Similarly, the GSN community standard says that a challenge to a claim, when supported by appropriate counter-evidence, "defeats" the claim [20, Clause 1:6.3.7]. The language from both documents goes further than suggesting the challenged claim is in a state of maximal uncertainty (i.e., zero confidence). Rather, raising a credible defeater against a claim shows the claim is in fact *false*.

It follows that, when reasoning quantitatively with defeaters, the term "confidence" is a misnomer. A more precise term is "belief" which can be used to describe the level of support for, or against, a claim. When represented on a numerical scale, one end denotes maximum belief in the claim's truth (i.e., certainty) and the other denotes maximum belief that the claim is false (i.e., rejection). The mid-point of the scale represents a state of maximum uncertainty in the claim's truth.

Taking the view that defeaters are negative claims representing points of logical inversion in an argument, symmetric terminology is used for defeaters. One end of the belief scale denotes a defeater as a well-founded challenge against a claim (i.e., certainty). The other end of the scale denotes a defeater as entirely unfounded or not credible (i.e., rejection). As with claims, the mid-point on the scale represents a state of maximum uncertainty in the defeater's credibility. When successfully rebutted with a counter-argument or counter-evidence, a defeater is sometimes referred to as "resolved" which is the same as saying it is rejected. If a defeater is not entirely rejected or resolved, then there is "residual" belief in the credibility of the defeater.

Through the remainder of this dissertation, the term "belief" is used to speak precisely about the truthfulness of assertions made in an AC. The term "confidence" is used to more generally refer to methodologies or activities for assessing the level of support in an AC's claims (e.g., the "the Bayesian Belief Network confidence assessment method").

4.3.2 Impact of Defeaters on Belief

When incorporating defeaters into a quantitative CAM using a belief scale as outlined above, it is important to decide how defeaters impact the belief in a claim in the argument. This dissertation identifies three possible interpretations of defeaters:

1) the *eliminative* interpretation[2], 2) the *deductive* interpretation, and 3) the *strict* interpretation. These interpretations are not mutually exclusive within an argument; for instance, a user of a CAM could decide to apply the eliminative interpretation in one sub-argument and the strict interpretation in another.

The Eliminative Interpretation

The eliminative interpretation says that "by eliminating defeaters we increase our [belief] in the argument's conclusion." (emphasis added) [31, p. 48]. That is, the work required to resolve a defeater gives additional credit towards belief in the claim. This interpretation is based on the notion of eliminative and defeasible reasoning, which underlies the Eliminative Argumentation method and its extension using Baconian Probabilities proposed by Goodenough et al. [31]. At the limit, a consequence of this interpretation is that it is possible to achieve maximum belief in a claim, even if no positive support is provided for it directly. Therefore, when using defeaters in this manner, it is important to ensure adequate support is provided for claims, in addition to showing that defeaters are resolved.

The Deductive Interpretation

The deductive interpretation says that the increase in belief in a parent claim is limited to the possible reduction that a defeater could have applied in the first place. Even if all defeaters are resolved, it is still possible to have uncertain belief in a claim if no positive support is provided for the claim. This is similar to how counter-examples are used in formal proofs: once a potential counter-example is shown to be invalid it no longer counts against the proof. This interpretation is consistent with the GSN Community Standard in regards to "counter" node types [20].

As an example, consider a claim c_0 that is challenged by defeater d_1 and otherwise unsupported by additional sub-claims or evidence. Suppose the belief in d_1 is initially determined to be *low* (i.e., slightly more positive belief than neutral). In the deductive interpretation, the belief in c_0 becomes slightly negative, skewing towards rejecting the claim as false. If d_1 is resolved, then the belief in claim c_1 returns to a neutral or uncertain state.

[2]In previous work [84] we referred to this as the *defeasible* interpretation; however, upon careful re-reading of the original work of Goodenough et al. it seems this is more aptly referred to as the *eliminative* interpretation [31, pp. 47-48].

The Strict Interpretation

The strict[3] interpretation says that if there are any unresolved defeaters against a claim, regardless of their degree of belief, they fully defeat the claim, even if the claim has alternative support. In some sense, the strict interpretation is an extreme version of the deductive interpretation. Some practitioners might favour a strict interpretation of defeaters, especially for extremely high-risk applications where the consequence of system failure can be catastrophic.

4.3.3 Requirements for Defeaters in Quantitative CAMs

Using the above principles, requirements for including defeaters in a quantitative CAM are derived. These requirements may be viewed as sub-requirements from Requirement R2. For brevity, these requirements assume a CAM is configured such that defeaters in the argument can reasonably impact the outcome of the assessment. It might be possible to tune parameters for a CAM such that belief in a claim or defeater is effectively ignored. For instance, a multiplicative weighting parameter could be set to zero such that belief in a defeater is zeroed-out in computations. Since the details of such parameters are method specific, it is not feasible to address all parameter configurations of all methods.

Requirements for Adding Defeaters

When a defeater is added, the belief in the parent claim should be reduced. However, the extent of reduction depends on many factors, such as the confidence in the defeater itself and relative weighting between the new defeater and other claims or defeaters that already exist.

R2-1 (Adding Credible Defeaters) *When a defeater is raised against a parent claim, provided there is some belief that the defeater is credible, the CAM shall decrease the belief in parent claim.*

R2-2 (Decomposing Defeaters) *The CAM shall permit the decomposition of defeaters into supporting "sub-defeaters" that refine the parent defeater's challenge against a claim.*

[3]This is a reference to using a compiler or transpiler in "strict mode" where compilation fails if any static analysis warnings are raised.

In the case of defeater decomposition, the child defeaters should not be thought of as challenging the veracity of the parent defeater. Rather, like belief propagates from a child claim to a parent claim, belief of defeaters should propagate from child defeaters to their parent. This requirement comes from practical experience applying dialectic reasoning, where it was found to be useful to organize groups of similar defeaters under a parent defeater to improve the readability of the argument [84]. Decomposition is used by other authors as well [31].

R2-3 (Weighting Defeaters) *The CAM shall provide a means to express the relative importance between defeaters that are challenging the same parent claim that is independent of the belief in the defeaters themselves.*

To see the importance of weighting defeaters, consider a scenario with two defeaters, both representing well-founded challenges to a claim; however, one defeater might be more important to the specific context of the claim and so its belief should be weighted more heavily than the other. The issue of relative importance is also raised by others [31, 80].

Requirements for Resolving Defeaters and Residual Defeaters

The change in belief due to resolving[4] a defeater depends on the interpretation of defeaters used, as described in Section 4.3.2.

R2-4 (Resolving Defeaters Increases Belief) *When a defeater challenging a parent claim is resolved, the belief in the parent claim shall stay the same or increase, but the nature of increase depends on the interpretation of defeaters applied for the argument step (eliminative, deductive, or strict).*

R2-5 (Resolving All Defeaters) *When a parent claim has defeaters challenging it, the CAM shall only output maximum belief in the parent claim if all defeaters are resolved.*

Note that, depending on the interpretation used, resolving all defeaters is necessary, but not sufficient, to achieve full belief in a claim.

R2-6 (Denoting Resolved Defeaters) *The CAM, when used in combination with a structured notation, should have a means to visually denote resolved defeaters.*

[4]In [84] we previously separated the cases where an "incredible" defeater was added and a defeater was resolved. However, both amounted to increases in the parent belief per R2-4 and so the redundant the requirement was removed.

For example, the GSN community standard includes provisions for marking a defeater as "defeated" (i.e., resolved) by crossing it out with an "X" symbol [20]. This is especially useful when presenting a structured argument to interest holders and discussing the role of defeaters in developing the argument.

R2-7 (Residual Defeaters) *For a residual defeater, the CAM shall provide the means to assign a belief value that can be propagated through the argument.*

The mechanism for assignment of belief to residual defeaters is specific to the CAM. Ideally, default values can be chosen for unresolved defeaters to reduce the amount of user input required by a user in the average case.

Requirements for Changes in the Belief of Children

The above requirements considered the addition and resolution of child defeaters challenging a parent claim. The following requirements consider the change in belief of a parent when the belief in a child claim or defeater changes.

R2-8 (Increasing Defeater Belief) *When a child defeater challenges a parent claim, if the belief in the defeater is increased, then the belief in the parent claim computed by the CAM shall decrease or stay the same.*

R2-9 (Decreasing Defeater Belief) *When a child defeater challenges a parent claim, if the belief in the defeater is decreased, then the belief in the parent claim computed by the CAM shall increase or stay the same.*

R2-10 (Increasing Claim Belief) *When a child claim (or other premise) rebuts a parent defeater, if the belief in the claim is increased, then the belief in the defeater computed by the CAM shall decrease or stay the same.*

R2-11 (Decreasing Claim Belief) *When a child claim (or other premise) rebuts a parent defeater, if the belief in the claim is decreased, then the belief in the defeater computed by the CAM shall increase or stay the same.*

For the above requirements, the magnitude of the change in belief will depend on several CAM-specific factors. For instance, the interpretation of defeaters used might impact how much belief in a parent claim changes in response to decreasing belief in a child defeater.

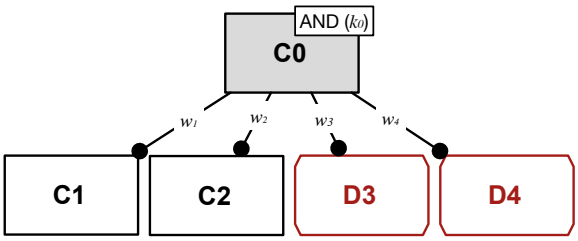


Figure 4.1: Running example for extending the BBN method with defeaters from [84].

4.3.4 Application of Dialectic Requirements

As a means of preliminary validation, the requirements for incorporating dialectic reasoning into quantitative CAMs are applied to extend the Bayesian Belief Network (BBN) method originally formulated by Hobbs and Lloyd [48]. The deductive interpretation of defeaters is used for this extension. First the extension of the BBN method to include defeaters is defined while showing how the requirements are satisfied. Then the extended BBN method is applied to an example from the automotive domain. The original BBN method is described in Chapter 2. As a running example, consider the small argument fragment (expressed using the EA notation) shown in Figure 4.1.

Adding Defeaters to the BBN Method

The BBN method represents belief as a subjective probability in $[0, 1]$. Using the definition of belief given above, 0 means the claim is rejected as false, 1 means we are certain the claim is true, and 0.5 is the point of maximum uncertainty. The BBN method uses a two-part approach to compute belief in a parent based on the belief of the children: 1) formulating a conditional probability table, and 2) computing belief in the parent node. Extending the BBN method to account for defeaters only requires modifying the CPT computation. There are four cases to consider.

Case 1: Parent is a Claim. If defeaters appear under a parent claim, as in Figure 4.1, then per R2-5, the parent is only true if all child defeaters are resolved. Effectively, a NoisyAND combinator is used to join the defeaters and then, either a NoisyAND or NoisyOR is used to join child claims. When defeaters are involved, the link parameter and leakage parameters allow the impact of the defeaters to be weighted in accordance with R2-3. The semantics of the weights differ slightly from Hobbs and Lloyd's original method. Using the nodes in Figure 4.1 as an example we have:

- **Link Parameter for Child Defeater (e.g., w_3)** - For a child defeater d_3 under a parent claim c_0 with a link parameter w_3, if d_3 is certainly true (i.e., $\Pr(d_3) = 1$), all other sibling defeaters are rejected as false (i.e., $\Pr(d_4) = 0$), and all sibling premises are certainly true (i.e., $\Pr(c_1) = \Pr(c_2) = 1$), then the belief in the parent is $\Pr(c_0) = 1 - w_3$. And similarly for d_4 with w_4.

- **Link Parameter for Child Claim (e.g., w_1)** - For the NoisyAND combinator, if c_1 was rejected as false, all other sibling claims are certainly true, and both d_3 and d_4 are resolved, then the belief in the parent claim is $\Pr(c_0) = 1 - w_1$. For the NoisyOR combinator, if all defeaters are resolved, c_1 is certainly true, and c_2 is rejected as false, then the belief in the parent is $\Pr(c_0) = w_1$. Likewise for c_2 with link parameter w_2.

- **Leakage Parameter (e.g., k_0)** - For the AND combinator, if all claims are certainly true and all defeaters are rejected as false, then the belief in the parent is $\Pr(c_0) = 1 - k_0$. For the OR combinator, if all claims are rejected as false and all defeaters are certainly true, then the belief in the parent is $\Pr(c_0) = k_0$.

Continuing the example in Figure 4.1, the CPT for a NoisyAND or NoisyOR among the claims is shown in Table 4.1. The revised CPT computation includes a multiplicative factor to account for the reduction in belief due to the defeaters. For the NoisyAND and NoisyOR combinators (among claims, respectively), each row in the CPT is computed as follows, where each $b_i \in 0, 1$:

$$\Pr(c_0 \mid x_1 = b_1, ...) = \left[(1 - k_0) \prod_{\substack{i=1, \\ x_i \text{ is C}}}^{n} (1 - w_i)^{(1-b_i)}\right] \cdot \prod_{\substack{j=1, \\ x_j \text{ is D}}}^{n} (1 - w_j)^{b_j} \tag{4.1}$$

$$\Pr(c_0 \mid x_1 = b_1, ...) = \left[1 - (1 - k_0) \prod_{\substack{i=1, \\ x_i \text{ is C}}}^{n} (1 - w_i)^{x_i}\right] \prod_{\substack{j=1, \\ x_j \text{ is D}}}^{n} (1 - w_j)^{b_j} \tag{4.2}$$

Case 2: Parent is a Defeater. There are two sub-cases to consider when the parent node is a defeater:

- **Case 2a: Rebutting by Claims** - If the children of a defeater are claims, then it is understood that the claims are rebutting the parent defeater. The belief in the child claim(s) must be combined and inverted such that increased belief in

Table 4.1: CPT for sub-argument from Figure 4.1 with $k_0 = 0.2$, $w_1 = 0.3$, $w_2 = 0.5$, $w_3 = 0.8$, and $w_4 = 0.7$.

#	$\Pr(c_1)$	$\Pr(c_2)$	$\Pr(d_3)$	$\Pr(d_4)$	$\Pr(c_0)$ **(AND)**	$\Pr(c_0)$ **(OR)**
0	0	0	0	0	0.28	0.20
...	...	...	...	...	...	...
12	1	1	0	0	0.80	0.73
13	1	1	0	1	0.24	0.22
14	1	1	1	0	0.16	0.14
15	1	1	1	1	0.05	0.04

the children results in a reduced belief for the parent defeater. Since inversion of belief is all that is necessary, it suffices to compute the CPT for the child claims as in the original BBN method and then invert the value by subtracting it from 1. For instance, for NoisyAND combinators: $\Pr(x_0 \mid b_1, ..., b_n) = 1 - \left[(1 - k_0) \prod_{i=1}^{n} (1 - w_i)^{1-b_i}\right]$.

- **Case 2b: Decomposition by Defeaters** - Instead of giving a counter argument against a defeater, an argument step might decompose a defeater into sub-defeaters (see R2-2). Then, the belief in the child defeaters must be propagated upward to form the belief in the parent. For instance, if the NoisyAND combinator is used, then belief in the parent defeater depends on belief among the child defeaters. Since this is simply propagating belief, the CPT for the step is computed as in the original method.

The case where the parent is a defeater and the children are a mix of claims and defeaters is not considered because it is not obvious how to interpret such an argument step using the available combinators from the BBN method.

Case 3: Defeater is a Leaf Node. If a defeater is the leaf node in the argument without any counter-argument against it, then it is considered "residual". Per R2-7, the user should assign a value representing the belief in the defeater. This is similar to how claim (or evidence) leaf nodes are handled by the original BBN method. If no belief assignment is given for a leaf defeater, then $\Pr(d_i) = 1.0$ is recommended as a conservative default. That is, in the absence of further information from the user, the defeater is taken to be certainly true. A less conservative, but perhaps more practical default might be $\Pr(d_i) = 0.5$ to indicate there is uncertainty about the defeater's credibility.

Case 4: Defeated Defeaters. R2-6 requires a provision for defeated defeaters. A defeated defeater is interpreted as one that is resolved by a suitably strong counter-argument. The defeated notation is an "override" that forces the defeater's belief to zero ($\Pr(d_i) = 0$) in the belief propagation algorithm.

Satisfaction of Requirements for Defeaters

Satisfaction of the above defeater requirements is argued by reasoning about the mathematics of the extended BBN method described above.

For R2-1, consider Equations 4.1 and 4.2 used to create the CPT. Defeaters are accounted for in these equations as an extra multiplicative factor: $\prod_{j \in D}(1 - w_j)^{b_j}$. A credible defeater causes $b_j = 1$ which means the $(1 - w_j)$ term is included to compute the belief of the parent claim. If $w_j > 0$, the belief in the parent is multiplied by a value in $(0, 1]$, producing a belief that is less than or equal to what the belief would have been without the defeater. For example, consider rows 12 and 13 in Table 4.1, that differ by the confidence of d_4 resulting in a reduced confidence for the parent claim. Applying similar rationale, R2-4 and R2-5 are satisfied.

R2-8 addresses the case where the belief in a defeater is increased. Consider the second step of the BBN method where the CPT is used to compute the parent node belief, the belief in a defeater is included in a sum of conditional probability expressions. Increasing the belief in a defeater weights the expression toward the rows in the CPT where the defeater is marked as certainly true, which have lower parent claim confidences by R2-1, thus lowering the belief in the parent claim. Similar rationale is applicable to R2-9, R2-10, and R2-11.

R2-7 is addressed by Case 3 above and R2-2 is addressed by Case 2b. R2-3 is addressed by the link parameters, which can be used to adjust the relative weight of defeaters. Finally, R2-6 is addressed by attaching semantics to the defeated notation described in Case 4.

Worked Example

To further validate the extended BBN method, it was applied to a fragment of an automotive AC for an adaptive cruise control (ACC) system[5,6]. The argument fragment considers the impact of the ACC system on the vehicle's control systems when it is

[5]This example is part of a larger AC for a fictitious, but representative, public ACC case study developed by Critical Systems Labs Inc.

[6]This example was modified from the original in [84] to improve clarity.

disengaged. The result is summarized in Figure 4.2. Beliefs are input at the leaves (E6112, E6121, and D6220) and automatically propagated in the argument using the specified link, leakage, and combinator parameters. Overall, the belief in the top-level claim of the argument fragment is $\Pr(c_{6000}) = 0.67$. Since 0.5 represents uncertainty in the claim's truth, this assessment is interpreted as moderate belief that claim C6000 is true. A justification for the chosen link, leakage, and combinator parameter values is provided below.

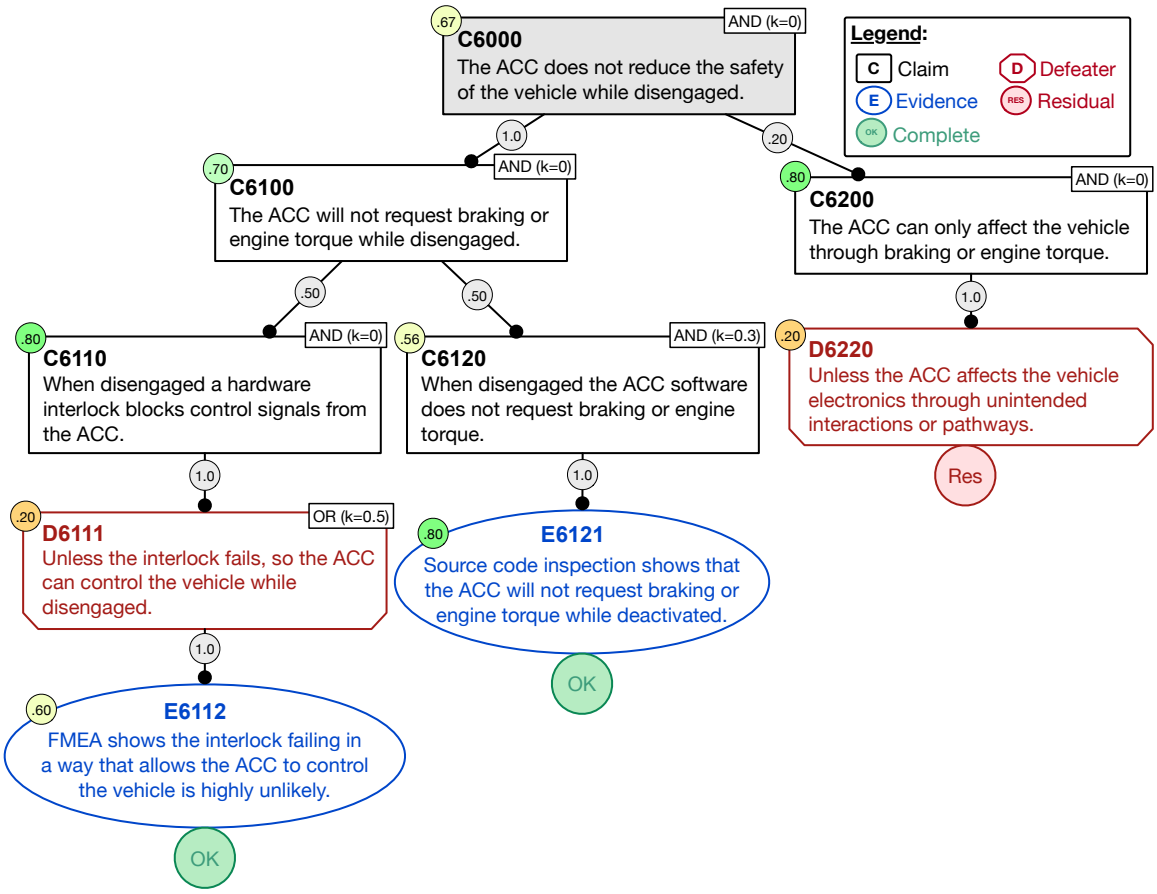


Figure 4.2: Worked example of the extended BBN method including defeaters.

C6000. For the top-level claim about the ACC not reducing vehicle-level safety while being disengaged to be true both children (C6100 and C6200) must be true, so a NoisyAND combinator is used with zero leakage. The link weight for child C6100 is chosen as $w_{6100} = 1.0$ because, even if C6200 was certainly true, the ACC requesting braking or engine torque while disengaged (i.e., C6100 is false) would be enough to entirely undermine the belief in C6000. On the other hand, the link weight for child

C6200 is only $w_{6200} = 0.20$ because C6200 being false only partially undermines C6000.

C6100. The support for the ACC not requesting braking or engine torque while disengaged in C6100 is split between two child claims describing two independent design measures: C6110 describes a hardware interlock and C6120 describes a property of the ACC software. Since both measures are depended upon equally, a NoisyAND combinator is used with zero leakage and evenly distributed link weights $w_{C6110} = w_{C6120} = 0.5$. That is, if only one of these measures was correctly implemented and the other was not at all implemented, then $\Pr(c_{6100}) \leq 0.5$.

C6110. The claim that the hardware interlock blocks control signals while the ACC is disengaged is challenged by a defeater suggesting that the interlock could fail (D6111). The case where a defeater challenges a parent claim was described above (see p. 104). The NoisyAND combinator is used with zero leakage. The link parameter $w_{6111} = 1.0$ implying that, if D6111 were certainly true, then the parent claim would be fully defeated.

D6111 and E6112. The defeater describing a potential interlock failure (D6111) is rebutted by evidence in the form of an FMEA showing the interlock is very unlikely to fail (E6112). The case where a child claim (or evidence) rebuts a defeater is described on page 105. The NoisyOR combinator with link parameter $w_{6112} = 1.0$ is used to capture the reasoning that if the FMEA results were entirely believable, then D6111 would be rejected as false. However, an FMEA is not on its own sufficient evidence to reject D111, so a leakage of $k_{6111} = 0.5$ is used to indicate that additional evidence is required to fully reject D6111. For example, suppose that the FMEA was of low quality, so only $\Pr(e_{6112}) = 0.60$ is assigned to the leaf node.

D6120 and E6121. The claim that the ACC's software does not request braking or engine torque while disengaged (C6120) is supported by inspection of the source code (E6121). However, source code inspection on its own is not enough to verify this behaviour of the software, so a NoisyAND combinator with leakage $k_{6120} = 0.3$ is used to show that, even if the source code inspection was perfect, belief in C6120 is limited to $\Pr(c_{6120}) \leq 1 - 0.3 \leq 0.7$. Finally, suppose the source code inspection was of reasonable (but imperfect) quality; so only $\Pr(e_{6121}) = 0.80$ is chosen.

C6200 and D6220. The claim that the only way the ACC can affect the vehicle is through braking and engine torque (C6200) is challenged by a defeater asking whether there are other (unknown) ways that the ACC could influence the vehicle (D6220).

No rebuttal is provided, and so a belief of $\Pr(d_{6220}) = 0.20$ is assigned to show that this defeater is believed to be unlikely, but cannot be entirely rejected as false. The link parameter $w_{6220} = 1.0$ is selected because, if `D6220` was certainly true, then it would be sufficient to fully defeat claim `C6200`.

Observations About the BBN Method

In extending the BBN method and applying it to the ACC example, two observations were made. First, unless the link weights and leakage parameters are carefully selected, the belief for the top claim in an argument is quite sensitive to the depth and breadth of the argument. Specifically, the probabilities diminish as the belief is propagated through the argument, even if the input beliefs at the leaves are relatively high. This is caused by multiplying many numbers in $[0, 1]$ together at each argument step, which inevitably produces smaller and smaller probabilities. This can lead to situations where the belief in the top claim does not match intuition; which is consistent with concerns raised by practitioners in Chapter 3.

Second, the semantics of the leakage and link parameters are difficult to apply. Interpreting the parameters involves making several (mental) negations, which is difficult in the context of all the other details of an argument. Additionally, the semantics change depending on whether an NoisyAND or NoisyOR combinator is used, which compounds the cognitive load required for the user.

4.3.5 Section Summary

This section has described a Dialectic Perspective on quantitative CAMs. To begin with, this section argued that the term "belief" is preferred to "confidence" when integrating dialectics into quantitative CAMs. Next, three interpretations of dialectic reasoning were defined: the eliminative, deductive, and strict interpretations. Requirements for integrating defeaters into quantitative CAMs were defined as a refinement of R2 from the Practitioner Perspective. Finally, as a means of preliminary validation, the requirements were used to extend the existing BBN quantitative CAM to incorporate defeaters. The extended method was then applied to an example from the automotive domain.

4.4 Chapter Summary

This chapter elicited requirements for a new confidence assessment method from three perspectives: the practitioner perspective, the dynamic perspective, and the dialectic perspective. As a means of preliminary validation for the requirements of the dialectic perspective, an existing CAM (the BBN method) was extended to account for dialectic reasoning. Subsequent chapters will use these requirements to develop and evaluate a new confidence assessment method.

Chapter 5
The *Certus* Language

The previous chapters established the need for alternative confidence assessment methods for dynamic ACs (DACs) that incorporate dialectic reasoning. This chapter develops the core of a new CAM called *Certus*, a domain-specific language for specifying belief assignment and propagation in DACs.

Certus is a mixed CAM, permitting both qualitative and quantitative reasoning about belief in DACs. Users assess belief using vague but linguistically meaningful expressions that are underpinned by a precise numerical interpretation provided by possibility and fuzzy set theory. The language's syntax is designed to be understandable, even for non-expert users. *Certus* can be used to reason about belief in arguments containing dialectic reasoning (i.e., "defeaters"). Moreover, to enable the assessment of DACs, the language permits the analysis of linked artifacts and indicators.

The chapter begins with a "first look" at *Certus* to orient the reader. Then prominent theories for managing imperfect knowledge are surveyed and evaluated against requirements from Chapter 4. Possibility and fuzzy set theory were selected and then used to construct a model of belief in claims for an AC. A denotational semantics for *Certus* is then specified, providing a precise interpretation of expressions authored in the language. The core of the language is then extended to permit reasoning about indicators and artifacts. Finally, an implementation of *Certus* is described, including its prototype integration into a commercial AC tool, *Socrates*.

5.1 A First Look at the *Certus* Language

Before diving into the detailed syntactic, semantic, and theoretical aspects of the *Certus* language, this section provides a brief orientation to the language by applying it to a small example. The example highlights key ideas underpinning the language, such as using linguistic valuations of belief (as opposed to strictly numerical valuations), and flexible and understandable belief propagation operators. The example also informally introduces core pieces of the language's syntax, including: belief assignment, belief propagation with the `cases` expression, and built-in macros.

Consider the argument in Figure 5.1, which is a simplified version of the adaptive cruise control (ACC) argument used in Chapter 4 and was previously published in [85]. Expressions in the *Certus* language are shown in partial rectangles connected to nodes in the argument via dashed lines[1]. The belief assessments annotate the top-right corner of each node. For example `C6100` is annotated by `L`, which corresponds to a *Low* level of belief that the claim `C6100` is true. Other annotations used in this example are: `SK` - *Skeptical*, `H` - *High*, `VH` - *Very High*, and `C` - *Certain*. See Table 5.3 for a list of pre-defined belief levels provided by the *Certus* language.

Beginning at the bottom of the left-most branch in Figure 5.1, the following paragraphs walk through the remainder of the example in detail, highlighting aspects of the *Certus* language along the way.

`C6110` —• `E6112`. A belief assignment is provided to the leaf node `E6112` about a Failure Modes and Effects Analysis (FMEA) that was performed on the hardware interlock controlling the ACC. Suppose this evidence was reviewed and determined to contain a high-quality analysis of potential interlock failures, i.e., we are "certain" (`C`) that the analysis was performed. Using *Certus*, the belief assignment is denoted: `E6112 is certain`. However, an FMEA is only one piece of evidence, which is not enough to fully support the claim about the hardware interlock. So, given the available evidence, we should limit the belief in `C6110` to at most `high`. This is accomplished by the belief propagation rule `C6110 is min(E6112, high)`.

`C6120` —• `E6121`. A belief assignment is provided to the leaf node `E6121` about source code inspection that was performed to show that the ACC's control software does not request braking or engine torque when disengaged. Suppose the quality of

[1]For the purpose of visualizing *Certus* expressions in this dissertation we use this notation; however, for a tool-based implementation one can imagine a less "crowded" user interface that displays this information as needed.

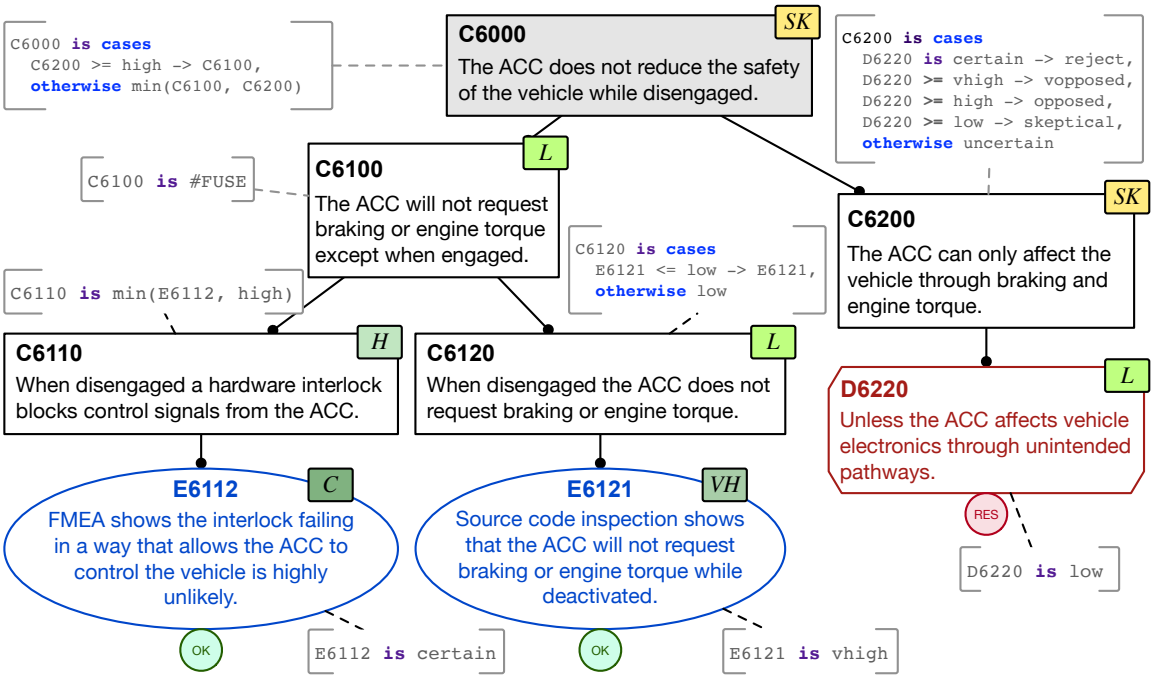


Figure 5.1: A small argument annotated with *Certus* expressions; modified from [85].

the code inspection was judged to be acceptable, but not perfect, so there is a very high level of belief in the evidence, which is captured by the expression `E6121 is vhigh`. However, code inspection results are not usually enough to support claims about software correctness, and so the propagation rule for `C6120` limits the credit that can be taken for the evidence. A `cases` expression is used to create a conditional statement over the belief in `E6121`. For `C6120`'s propagation rule, only one case is used: if the condition of the case (the Boolean expression before the `->`), then the corresponding value (the right-hand side of the `->`) is taken as the output of the `cases` expression and assigned to `C6120`. In this example, the output is not a named belief level, but rather whatever belief was assigned for `E6121`. Finally, if no cases are matched, then the `otherwise` case is used. Interestingly, this propagation rule is equivalent to the expression: `C6120 is min(E6121, low)`.

`C6100` —• `C6110, C6120`. Belief in the claim `C6100` depends on the belief in two child claims, `C6110` and `C6120`, with both children describing complementary measures for ensuring the ACC does not impact the vehicle's motion when disengaged. If we believed these mechanisms were truly independent, then we could model this situation using an expression like `C6100 is max(C6110, C6120)`. However, for the purpose of introducing another facet of the *Certus* language, suppose these mechanisms are not

entirely independent and that belief in C6100 is more accurately described as the average belief among the children. To model this relationship, the #FUSE macro[2] is used. Similar to general purpose programming languages, macros in *Certus* are defined as shortcuts that can be expanded to full expressions in the language. The expansion of macros is sensitive to the number and type of child nodes. In this example, the #FUSE macro is expanded to a cases expression that matches all possible combinations of pre-defined belief levels on the child claims C6110 and C6120 and then outputs a belief level that is between the two child levels, conservatively "rounded down". As depicted in Figure 5.1, this produces L(ow) belief for C6100.

C6200 —• D6220. A belief assignment is provided for the leaf defeater D6220 describing the possibility that the ACC can influence other aspects of the vehicle, other than by issuing commands for braking or engine torque[3]. Suppose we have some reason to believe this is a weakly credible defeater and so it is assigned a Low level of belief. Additionally, the parent claim C6200 does not have any evidence provided in support of it. Applying the deductive interpretation of defeaters from Section 4.3.2 means that the maximum level of belief in C6200 can be *uncertain*. With this in mind, the propagation rule for C6200 is given as a cases expression that inverts the belief level from the defeater to produce the belief in the parent, up to uncertainty in the parent. cases expressions are evaluated in the order they are written, and the first matching condition is D6220 >= low, so C6200 is assigned the belief level SKeptical.

C6000 —• C6100,C6200. Finally, the belief in the parent claim C6000 is computed with a cases expression that encodes the following reasoning. If there is sufficient belief (i.e., at least High belief) that the ACC only influences the vehicle through braking and engine torque (C6200), then we accept the belief in C6100 directly. Otherwise, the lowest belief among the children is used. Overall, there is skepticism about whether C6000 is true; that is, there is slight belief that C6000 is false. Examining the totality of evidence supporting this argument fragment, this is not a surprising outcome. Though each piece of evidence presented in support of C6110 and C6120 was acceptable, it is not enough to fully support these claims. Moreover, there is no evidence presented in support of C6200, but there is a reason to (weakly) doubt its truth.

[2]To avoid confusion with the well-established concept of an arithmetic mean, in *Certus* we call this "fuse" (or "fusion") because it is not computed exactly like an average.

[3]There is an entire field of research focused on developing methods for identifying unexpected "feature interactions" in complex systems.

5.2 Theories for Imperfect Knowledge

ACs, like many real-world engineering work products, depend on *imperfect knowledge* about a system and its environment. System developers often reason about uncertain, incomplete, or even contradictory information. For instance, a design specification might be incomplete, a technical analysis might have limitations, or verification activities might yield conflicting results. In day-to-day practice, system developers rely on experience and intuition[4] to navigate these situations.

The first step towards formally defining the *Certus* language is to select a theory for representing and reasoning about imperfect knowledge [179]. Several major theories for reasoning about imperfect knowledge are introduced below, including foundational notation and definitions[5]. Then the theories are considered in the context of the requirements from Chapter 4. The theories reviewed in this section were chosen based on their use in the CAM literature as a basis of quantitative confidence assessments, and based on the intersection of theories discussed in textbooks on managing imperfect knowledge [142, 143].

5.2.1 Theories for Reasoning About Imperfect Knowledge

Many mathematical theories exist for reasoning about imperfect knowledge. Five prominent examples are probability theory, subjective logic (SL), Dempster-Shafer Theory (DST), possibility theory, and fuzzy set theory. Some of these theories have already been used as the basis for various CAMs. For example, Idmessaoud's method is based on DST [50], Herd et al. used subjective logic [154], and the BBN method is founded on probability theory [48].

Probability Theory

Probability theory is the most well-known theory for modelling imperfect knowledge with a very wide range of applications. There are two main perspectives one can take when considering a probability assignment [142]. The frequentist interpretation views probabilities as representations of empirical observations of experiments (e.g., flipping a fair coin 100 times should yield approximately 50 tails). Alternatively,

[4]This is sometimes called "engineering judgement".

[5]Notation and definitions of Possibility Theory and Fuzzy Set Theory are introduced in detail as these will be used later for reasons discussed below. Other theories are introduced with less detail for brevity.

the Bayesian interpretation views probability as describing the likelihood of an event occurring based on prior knowledge or subjective belief [180].

Regardless of the interpretation, the fundamental axioms of probability (also called the Kolmogorov Axioms) apply [142, 181]. Given two events or outcomes x and y in a universe of potential outcomes Ω, the following must be true: 1) $\Pr(x) \in [0, 1]$; 2) $\Pr(x) = 1$ indicates outcome x certainly occurs; and 3) $\Pr(x \cup y) = \Pr(x) + \Pr(y)$, for disjoint outcomes x and y.

An important consequence of these axioms is the law of total probability which states the sum of probabilities of all outcomes must equal 1, i.e., $\sum_{x \in \Omega} \Pr(x) = 1$ [181]. Additionally, if two outcomes are independent, then the product of their probabilities gives the probability that both outcomes, i.e., $\Pr(x \cap y) = \Pr(x) \cdot \Pr(y)$.

Representing Imperfect Knowledge with Probability. A probability measure can be taken as a degree of *uncertainty* about the occurrence of an outcome. For instance, a probability of 0.75 suggests the outcome is more likely than not to occur; and a probability of 0.05 means an outcome is quite unlikely to occur. Both of these assertions describe a state of knowledge about the world, even if an outcome occurring is uncertain. However, it is not possible to fully describe a state of *ignorance* (i.e., "I don't know") in probability theory in a single probability measure for an outcome [179]. To express ignorance a uniform probability distribution must be used, describing the probabilities of multiple outcomes.

Subjective Logic

Subjective Logic uses probabilistic logic to manipulate belief about the occurrence of outcomes or the truth of propositions [151, 154]. Opinions are used to represent belief about an outcome and are modelled as 4-tuples. For simplicity, consider a binary universe with two outcomes $\Omega = \{x, \bar{x}\}$. A binomial opinion, $\omega_x = (b, d, u, a)$ with b representing the *belief* mass in favour of x occurring, d representing the *disbelief* mass against x occurring, u representing *uncommitted* belief mass (i.e., uncertainty), and a representing the *base rate* uncertainty, in the absence of any additional information, usually taken as $a = 0.5$ for binomial opinions. Additionally, it is required that the total belief mass sum to 1, i.e., $\sum_{x \in \Omega} b_x = 1$. The opinion 4-tuple can be converted to a corresponding Beta probability distribution that describes the probability mass for the truth of the claim.

Subjective Logic provides a number of operators for manipulating opinions. One

example is the conditional deduction operator, which allows an opinion about an outcome, y, to depend on the opinion about another outcome x [154].

Dempster-Shafer Theory

Dempster-Shafer Theory (DST), also called "evidence theory", is a mathematical framework for reasoning about uncertainty [141] that generalizes probability theory using a *belief* and *plausibility* measure [142, 143].

Given a universe of possible outcomes, $\Omega = \{x_1, ..., x_n\}$, a mass function associates a set of outcomes with a degree of belief, $m : \mathcal{P}(\Omega) \rightarrow [0, 1]$, where $m(x) = 0$ indicates complete disbelief and $m(x) = 1$ indicates certainty. The mass function is sometimes referred to as a basic belief function or belief assignment. A valid mass function satisfies the following constraints: 1) $m(\emptyset) = 0$, i.e., a non-outcome cannot be observed; and 2) $\sum_{A \in \mathcal{P}(\Omega)} m(A) = 1$, at least *some* outcome must occur [142]. A consequence of (2) is that there is a fixed amount of belief mass that can be distributed among outcomes, and that the universe of outcomes is a closed-world [182, p. 42].

Using the mass function, a belief measure, $\text{Bel} : \mathcal{P}(\Omega) \rightarrow [0, 1]$, and plausibility measure, $\text{Pl} : \mathcal{P}(\Omega) \rightarrow [0, 1]$, are defined for a subset of outcomes, $A \in \mathcal{P}(\Omega)$:

$$\text{Bel}(A) = \sum_{A' : A' \subseteq A} m(A') \quad \text{and} \quad \text{Pl}(A) = \sum_{A' : A' \cap A \neq \emptyset} m(A') \tag{5.1}$$

Given a subset of outcomes $A \in \mathcal{P}(\Omega)$, the belief and plausibility measures can be seen as bounding the probability of an outcome in A occurring, i.e., $\text{Bel}(A) \leq \text{Pr}(A) \leq \text{Pl}(A)$. Additionally, the belief and plausibility measures are duals such that $\text{Pl}(A) = 1 - \text{Bel}(A^C)$. The degree of uncertainty can be computed as $\text{Uncert}(A) = 1 - \text{Bel}(A) - \text{Disb}(A)$, and the degree of disbelief as $\text{Disb}(A) = \text{Bel}(A^C)$.

DST mass functions are combined using *Dempster's Rule of Combination*, denoted $\oplus$, to merge information obtained from different sources. Intuitively, the rule combines belief from shared sources of evidence and ignores conflicting information. Given two mass functions m_1 and m_2 over their respective universes of observations U_1 and U_2, the combined mass function is [142]:

$$(m_1 \oplus m_2)(A) = \sum_{U_1, U_2 : U_1 \cap U_2 = A} \frac{m_1(U_1) \cdot m_2(U_2)}{\left[\sum_{U_x, U_y : U_x \cap U_y \neq \emptyset} m_1(U_x) \cdot m_2(U_y) \right]} \tag{5.2}$$

Fuzzy Set Theory

The theory of "fuzzy" sets was developed by Zadeh in the 1960s [183] and has been widely used in a range of engineering and scientific applications [143]. It also provides a basis for fuzzy logic [184] and possibilistic logic (see Section 5.2.1).

An important characteristic of fuzzy sets and fuzzy logic is that they allow for the expression of *imprecise* or *vague* knowledge. There are many examples where imprecise knowledge is used, including in natural language. For instance, the statement "Sam is tall" does not provide a precise definition of "tall". Does this mean that Sam is taller than the average person of their age? Or, does it mean that their height has exceeded a threshold beyond which all people are considered tall? Or something else? Despite the imprecision in the statement, many people who know Sam might readily agree that they are tall, even without knowing Sam's exact height. That is, natural language provides a means to reason vaguely, without knowing exact details. Fuzzy sets can be used to formalize such expressions, while maintaining the utility gained by vague reasoning.

Given a universe of discourse, $\Omega = \{x_1, x_2, ..., x_n\}$, the membership of elements in a classical (also called "crisp") set, A, is binary: either $x \in A$ or $x \notin A$. Fuzzy sets express membership in degrees, i.e., an element x is in a fuzzy set $\underset{\sim}{B}$ to some extent. The extent of membership is given by a membership function $\mu_B : \Omega \rightarrow [0, 1]$ where $\mu_B(x) = 1$ when x is firmly in the set $\underset{\sim}{B}$ and $\mu_{\underset{\sim}{B}}(x) = 0$ when x is fully outside $\underset{\sim}{B}$. We denote the set of all fuzzy sets over universe Ω as $\mathcal{F}_\Omega$, or simply $\mathcal{F}$ if the domain is obvious.

Figure 5.2 illustrates the difference between a crisp and a fuzzy set. In both cases, x_1 is not in the set and x_2 is in the set. However, x_3 is only partially a member of fuzzy set $\underset{\sim}{B}$. Using Zadeh's notation [183], the membership function $\mu_{\underset{\sim}{B}}$ for this discrete universe consisting of $\{x_1, x_2, x_3\}$, might be $\mu_{\underset{\sim}{B}} = \left\{0/x_1,\ 1/x_2,\ 0.6/x_3\right\}$.

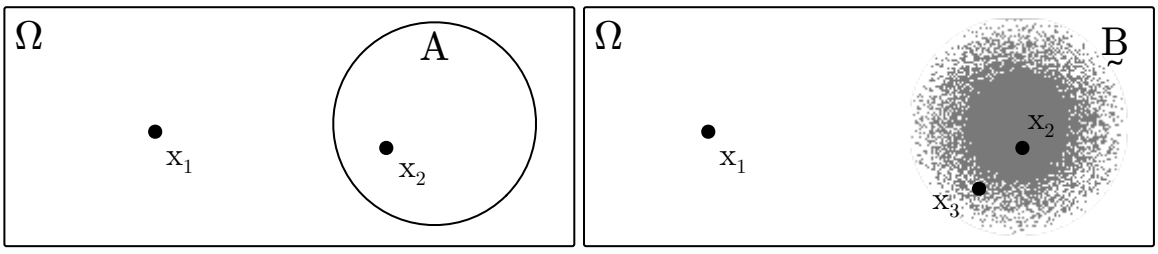


Figure 5.2: Visualization of a crisp (A) and fuzzy set ($\underset{\sim}{B}$) over a universe Ω.

When Ω is a continuous range, e.g., $[0, 1]$, the membership function of a fuzzy

set is visualized on a 2-dimensional plot where the value being evaluated is on the horizontal axis and the degree of membership is shown on the vertical axis. See Figure 5.3 for an example that corresponds to the crisp set $A = [0, 1]$ and the fuzzy set $\underset{\sim}{B}$ from Figure 5.2. Note that the membership of x_3 in $\mu_{\underset{\sim}{B}}$ is between 0 and 1 indicating the x_3 is only in $\underset{\sim}{B}$ to some degree, $\mu_{\underset{\sim}{B}}(x_3) = 0.6$.

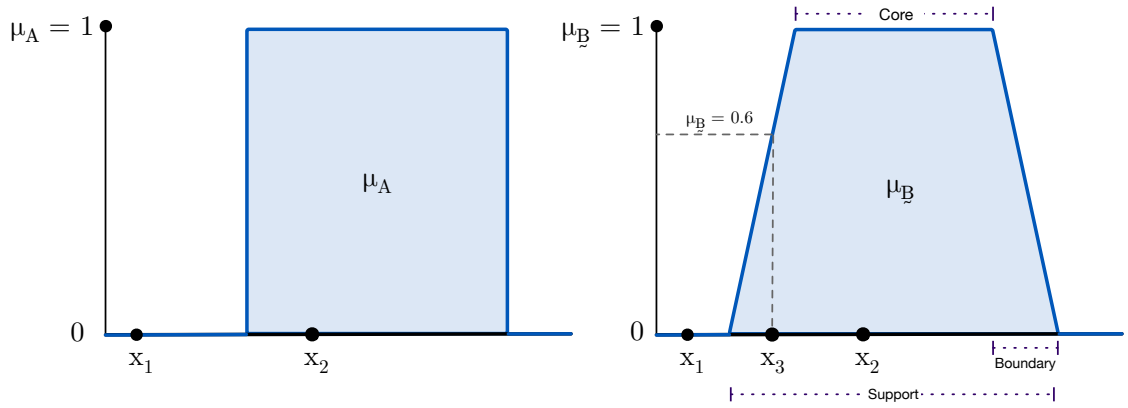


Figure 5.3: Examples of crisp and fuzzy membership functions for sets A and $\underset{\sim}{B}$.

Several definitions relating to the shape of a fuzzy set's membership function are relevant [143, pp. 90-92]. The right-hand membership function in Figure 5.3 labels the core, support, and (right-side) boundary for $\underset{\sim}{B}$ and is both a normal and a convex membership function.

Definition 6 (Core of a Fuzzy Set) *The **core** of a fuzzy set $\underset{\sim}{A}$ is the region of the universe where the membership function assigns full membership to elements in the universe, i.e., the core is defined by the set* $\left\{x : x \in \Omega,\ \mu_{\underset{\sim}{A}}(x) = 1\right\}$.

Definition 7 (Support of a Fuzzy Set) *The **support** of a fuzzy set $\underset{\sim}{A}$ is the region of the universe where the membership function assigns at least partial membership to elements in the universe, i.e., the support is defined by the set* $\left\{x : x \in \Omega,\ \mu_{\underset{\sim}{A}}(x) > 0\right\}$.

Definition 8 (Boundary of a Fuzzy Set) *The **boundary** of a fuzzy set $\underset{\sim}{A}$ is the region of the universe where the membership function assigns strictly partial membership to elements in the universe, i.e., the boundary is defined by the set* $\left\{x : x \in \Omega,\ \mu_{\underset{\sim}{A}}(x) > 0 \wedge \mu_{\underset{\sim}{A}}(x) < 1\right\}$.

Definition 9 (Normal Fuzzy Set) *A fuzzy set $\underset{\sim}{A}$ is **normal** if its membership function assigns full membership to at least one element of the universe, i.e.,* $\exists x \in \Omega : \mu_{\underset{\sim}{A}}(x) = 1$. *A fuzzy set that is not normal is called **subnormal**.*

Definition 10 (Height of a Fuzzy Set) *The **height** of a fuzzy set $\underset{\sim}{A}$ is defined as the maximum value assigned by the membership function, i.e., $h(\underset{\sim}{A}) = \max_{x \in \Omega} \mu_{\underset{\sim}{A}}(x)$.*

Definition 11 (Convex Fuzzy Set) *A fuzzy set $\underset{\sim}{A}$ is **convex** if its membership function's values are: i) strictly monotonically increasing, ii) strictly monotonically decreasing, or iii) strictly monotonically increasing and then monotonically decreasing. Formally, for any choice of $x, y, z \in \Omega$ such that $x < y < z$, then $\mu_{\underset{\sim}{A}}(y) \geq \min(\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{A}}(z))$.*

Operations on Fuzzy Sets. Just as there are complement, union, and intersection operators on crisp sets, there are similar operations on fuzzy sets. These operations act on the fuzzy membership functions of the sets to produce a new membership function [143, p. 35]:

Definition 12 (Fuzzy Set Complement) *Given a fuzzy set $\underset{\sim}{A}$, the **complement** is defined as $\mu_{\underset{\sim}{A}^C}(x) = 1 - \mu_{\underset{\sim}{A}}(x)$.*

Definition 13 (Fuzzy Set Inversion) *Given a fuzzy set $\underset{\sim}{A}$ defined over $[0, 1]$, the **inversion** (i.e., "mirror image") is defined as $\mu_{\underset{\sim}{A}^I}(x) = \mu_{\underset{\sim}{A}}(1 - x)$.*

Definition 14 (Fuzzy Set Union) *Given fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$, the **union** of the fuzzy sets is defined as $\mu_{\underset{\sim}{A} \cup \underset{\sim}{B}}(x) = \max(\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x))$.*

Definition 15 (Fuzzy Set Intersection) *Given fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$, the **intersection** of the fuzzy sets is defined as $\mu_{\underset{\sim}{A} \cap \underset{\sim}{B}} = \min(\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x))$.*

As with crisp sets, various containment operators exist for fuzzy sets. In addition to the usual subset operator [143, p. 35], we introduce a similar "overlap" operator that will be useful later.

Definition 16 (Fuzzy Subset) *Given fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$, $\underset{\sim}{A}$ is a **subset** of $\underset{\sim}{B}$ if $\forall x \in \Omega : \mu_{\underset{\sim}{A}}(x) \leq \mu_{\underset{\sim}{B}}(x)$.*

Definition 17 (Fuzzy Overlap) *Given fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$, $\underset{\sim}{A}$ **overlaps** $\underset{\sim}{B}$ if they have overlapping support, i.e., $\exists x \in \Omega : \mu_{\underset{\sim}{A}}(x) > 0 \wedge \mu_{\underset{\sim}{B}}(x) > 0$.*

The α-cut is an important operation for transforming a fuzzy set into a crisp set, this is sometimes called "defuzzification". The α-cut generates a new (crisp) set containing only elements whose membership is at least α [143, p. 95].

Definition 18 (α-cut) *Given a fuzzy set $\underset{\sim}{A}$ and membership degree α, the α-cut*

produces a set A_α such that:

$$\mu_{A_\alpha}(x) = \begin{cases} 1 & \text{if } \mu_{\underset{\sim}{A}}(x) \geq \alpha \\ 0 & \text{otherwise} \end{cases}$$

Other methods for defuzzification exist for computing scalar representations of sets, such as computing the centroid of a set [143, p. 98], which returns a single number representing the set, which is similar to computing an average.

Ranking Fuzzy Sets. It is sometimes useful to order fuzzy sets based on their position (and mass distribution) along their domain. Special operations are required to *rank*[6] fuzzy sets. With a ranking function, additional operators like $<$ and $>$ can be defined. Several procedures for ranking fuzzy sets have been proposed [185]. Among the available procedures, Yager's F measure[7] is selected because it yields correct results for convex and normalized membership functions and is relatively simple [186, 185].

Definition 19 (Yager's Ranking Function) *Yager's ranking function, $F : \mathcal{F}_{[0,1]} \rightarrow [0, 1]$, computes the integral of the mean of level sets to produce a number that represents the position (i.e., the rank) of the fuzzy set in $[0, 1]$.*

$$F(\underset{\sim}{A}) = \int_0^{h(\underset{\sim}{A})} M(A_\alpha)\, d\alpha$$

Where A_α is the α-cut of the set $\underset{\sim}{A}$ (i.e., a "level set") and $M(A_\alpha)$ is the mean of the values within the level set.

Comparing the values produced by Yager's ranking function for different fuzzy sets allows one to order the sets, i.e., to declare one set "greater than" another.

Representing Vague Knowledge The ability to represent vague knowledge is an important capability for fuzzy sets. Using the example of Sam's body height introduced above, we can define several fuzzy membership functions that connect words in English with the degree to which they represent concepts relevant to body height. Different natural language characterizations for describing the height of adult hu-

[6]Defining a ranking operation is a surprising difficult problem due to the variety of shapes that a fuzzy membership functions can assume. Interestingly, Bortolan and Degani compared several ranking operations and did not find any ideal ranking procedure among the cases they studied, and even found cases where the intuition about how to order fuzzy sets was not obvious [185].

[7]Yager proposed three versions of this measure, F_1, F_2, and F_3, we use F_3 from [186].

mans are *short*, *average*, *tall*, *Basketball* (Star). These are interpreted as the fuzzy set membership functions are shown in Figure 5.4 below.

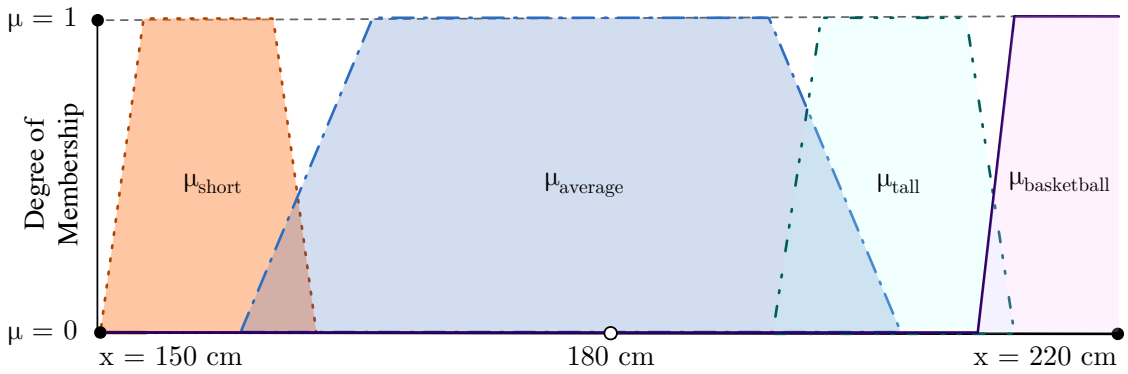


Figure 5.4: Examples of encoding vague knowledge as fuzzy set membership functions over the domain of adult human body heights.

Common Fuzzy Membership Functions There are several common membership functions that are regularly used to construct fuzzy sets. These are depicted in Figure 5.5. For parameters $a, b, c, d \in [0, 1]$ (or otherwise in the domain of the fuzzy set) with $a \leq b \leq c \leq d$, these are defined as piece-wise functions:

- $\mu_{\text{down}}(x) = \left\{ x < a \rightarrow 1,\ a \leq x \leq b \rightarrow \frac{x}{(a-b)},\ x > b \rightarrow 0. \right.$
- $\mu_{\text{up}}(x) = \left\{ x < a \rightarrow 0,\ a \leq x \leq b \rightarrow \frac{(x-a)}{(b-a)},\ x > b \rightarrow 1. \right.$
- $\mu_{\text{trap}}(x) = \left\{ a \leq x \leq b \rightarrow \frac{(x-a)}{(b-a)},\ b < x < c \rightarrow 1,\ c \leq x \leq d \rightarrow \frac{x}{c-d},\ \text{o/w} \rightarrow 0. \right.$
- $\mu_{\text{tri}}(x) = \left\{ a \leq x \leq b \rightarrow \frac{(x-a)}{(b-a)},\ b \leq x \leq c \rightarrow \frac{x}{c-d},\ \text{o/w} \rightarrow 0. \right.$

For brevity, we sometimes show these functions with the parameters a, b, c, d as arguments, e.g., $\text{trap}(a, b, c, d)$. This should be understood as a higher-order function that instantiates a specific membership function (as defined above) over the relevant domain.

Other common membership functions include stem functions (a single value in the domain is in the set, all others are out), constant functions (all elements in the domain have the same degree of membership), sigmoid functions, and Gaussian functions.

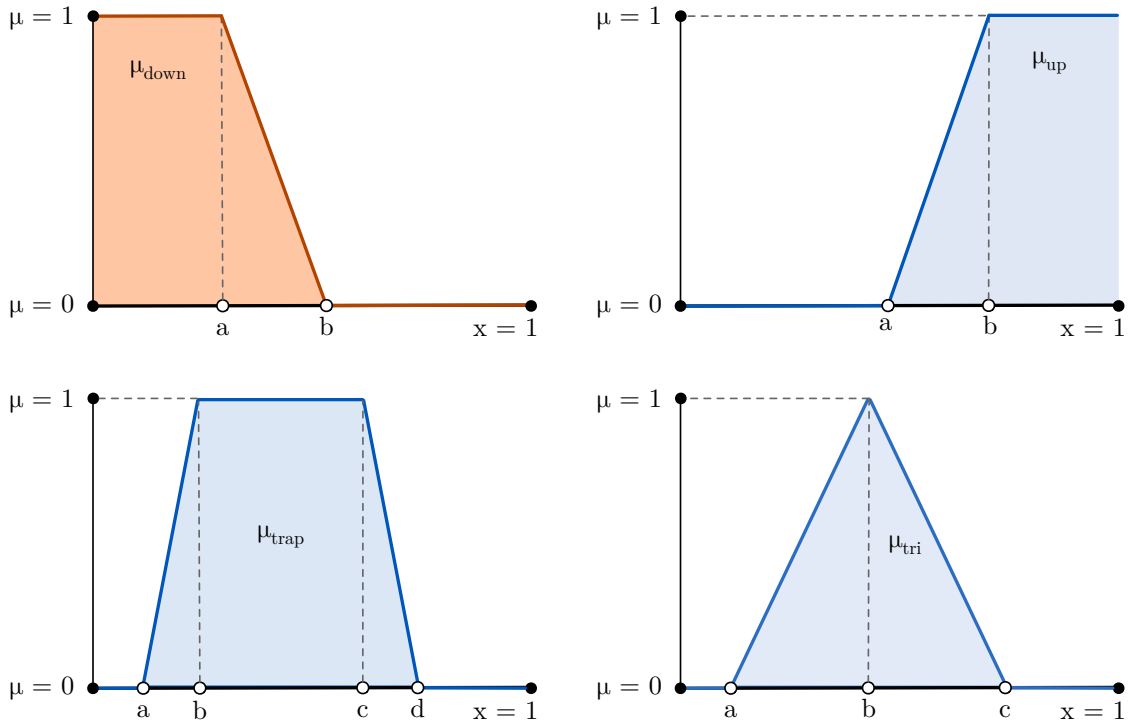


Figure 5.5: Common fuzzy set membership functions.

Possibility Theory

Possibility theory and possibilistic logic provide a mathematical framework for reasoning over uncertain, incomplete, and partially inconsistent knowledge. The concept was first proposed by Zadeh [187] and then expanded upon by Dubois and Prade [188, 189, 190]. From the beginning, possibility theory has been linked with the notion of a fuzzy set, with membership functions being synonymous with possibility distributions [187].

Consider a finite universe of potential outcomes, $\Omega = \{x_1, x_2, ..., x_n\}$. A possibility distribution, $\pi : \Omega \rightarrow [0, 1]$, describes the degree to which an outcome in the universe is possible. If $\pi(x) = 0$, then the outcome is rejected as impossible and will not occur. If $\pi(x) = 1$, then the outcome is entirely plausible, and it would not be surprising if it occurs [190].

Using the possibility distribution, two measures are defined over sets of outcomes. The *possibility* measure, $\Pi : \mathcal{P}(\Omega) \rightarrow [0, 1]$, selects the most plausible outcome from a set of outcomes. That is, the possibility measure gives the degree to which the available evidence does not contradict an outcome. The *necessity* measure,

$N : \mathcal{P}(\Omega) \rightarrow [0, 1]$ describes the extent to which an outcome is required to occur. That is, the necessity measure gives the degree to which available evidence supports an outcome occurring. For a set $A \in \mathcal{P}(\Omega)$, these measures are defined as duals:

$$\Pi(A) = \max_{x \in A} \pi(x) \quad \text{and} \quad N(A) = 1 - \Pi(A^C) \tag{5.3}$$

The values of the possibility measure, Π, are interpreted as described above for π. For the necessity measure, $N(A) = 0$ means that the outcomes in A are not at all necessary or required to occur, and $N(A) = 1$ means that an outcome in A is necessary, and will certainly occur.

Possibility Distributions as Fuzzy Sets. A possibility distribution, π, can be viewed as the membership function, μ, of a fuzzy set [190]. For example, consider the expression "Sam is tall", with a fuzzy set μ_{tall} defined over the possible heights for human bodies in centimeters. The degree of membership in the fuzzy set also describes the possibility of a specific outcome. If $\mu_{\text{tall}} = \{..., 0.97/201, 1/202, 0.98/203, ...\}$, then given the vague information that "Sam is tall" (which is represented by μ_{tall}), the possibility of the assertion "Sam is 201 centimeters tall" would be $\pi_{\text{tall}}(201) = 0.97$. Using the duality relationship above from Equation 5.3, a necessity measure could also be computed for this assertion.

This correspondence between fuzzy sets and possibility distributions is described because it was central to Zadeh's original formulation of possibility theory [187] and to show possibility theory's ability to reason over vague knowledge. Later in this chapter both fuzzy and possibility distributions are used, but this specific correspondence between the theories is not exploited.

Operators on Possibility Measures. Since the possibility and necessity measures are defined over sets, it is natural to also define corresponding set operations, such as union and intersection. For two sets of outcomes $A, B \in \Omega$, the following hold[190]:

$$\Pi(A \cup B) = \max\Big(\Pi(A), \Pi(B)\Big) \quad \text{and} \quad N(A \cap B) = \min\Big(N(A), N(B)\Big) \tag{5.4}$$

Notably, possibility theory does not define the usual arithmetic operators, such as $+$ or $\cdot$, over the possibility and necessity measures. Instead, min and max are the main operations used to manipulate these measures. A benefit of this is that possibility theory is computationally simple, compared to other theories.

Possibilistic Logic. Building on the possibility and necessity measures defined above, possibilistic logic permits for reasoning over the possibility and necessity of expressions. A popular formulation is to use a necessity-valued logic, where formulas in propositional or first-order logic are paired with a necessity valuation representing a measure of certainty in the truth of the formula. This is expressed as a pair $\langle \phi, (N\ \alpha) \rangle$ where ϕ is a formula and $(N\ \alpha)$ is its necessity such that $N(\phi) \geq \alpha$ [189]. The usual logical reasoning rules are extended to include necessity valuations to facilitate logical inference. For instance, the well-known *modus ponens* (MP) rule is generalized to [189]:

$$\langle \phi_1, (N\ \alpha) \rangle,\ \langle \phi_1 \rightarrow \phi_2, (N\ \beta)\ \rangle \vdash \langle \phi_2, \min(\alpha, \beta) \rangle \tag{5.5}$$

Representing Ignorance. An important feature of possibility theory and logic is its ability to clearly express *ignorance*. This feature arises from having two measures, possibility and necessity, that describe the state of knowledge about an outcome. A state of ignorance is represented by $\Pi(x) = 1$ and $N(x) = 0$, where outcome x is entirely possible because there is no reason to doubt it, but where there is also no support for x. Our knowledge about the outcome is "incomplete" in the sense that we have nothing to suggest it will occur, or not.

Representing Contradicting Knowledge. Another important feature of possibility theory is that it can express partially *inconsistent* or *contradicting* knowledge, where evidence is presented in favour of an outcome and also against the outcome [188]. This feature arises from permitting non-normal possibility distributions (i.e., $\max_{x \in \Omega} \pi(x) < 1$). The consistency measure of a universe is the maximum possibility among all outcomes in the universe, $\text{Cons} = \max_{x \in \Omega} \pi(x)$. Then the inconsistency is $\text{Incons} = 1 - \text{Cons}$. The notion of consistency is extended for sets of formulas in possibilistic logic by considering satisfying models of the set of formulas.

As an example, consider a universe with two outcomes $\Omega = \{x_1, x_2\}$ and a possibility distribution with $\pi(x_1) = 0.3$ and $\pi(x_2) = 0.85$. This distribution is inconsistent with $\text{Incons} = 1 - 0.85 = 0.15$. Computing the possibility and necessity measures for x_1 gives $\Pi(\{x_1\}) = 0.3$ and $N(\{x_1\}) = 1 - 0.85 = 0.15$. These measures are in some degree of conflict: outcome x_1 is far from possible (in fact, it is closer to impossible than possible), yet there is still a low degree of certainty that it will occur due to the necessity measure.

5.2.2 Evaluation of Theories

Chapter 4 identified several requirements for a new CAM. The ease of satisfying the requirements depends on the choice of theory for managing imperfect knowledge. Therefore, as part of developing a new CAM, it is necessary to analyze the theories in the context of these requirements.

Not all requirements from Chapter 4 are applicable to this stage of method development, or would not offer much discriminating power among the theories for managing imperfect knowledge. For example, requirement R1 (Structured Argumentation) is trivially satisfied by any CAM that can operate over a structured argument, regardless of underlying theory. As another example, requirement R8 (Stopping Criteria) is more relevant when defining the procedure(s) for the CAM than when selecting a foundational theory for the method. The remainder of this sub-section evaluates each of the theories above in the context of five relevant CAM requirements.

R2 - Defeaters

Requirement R2 demands that a CAM support dialectic reasoning (i.e., "defeaters"). Generally speaking, defeaters represent points where belief is inverted in an AC's argument [84], though the specific interpretation of how defeaters impact belief depends on how they are incorporated. Regardless, for a theory to support defeaters it should define a measure over which belief in claims can be interpreted. For instance, per requirement R2-1, the existence of a credible defeater should result in some decrease in belief, which implies the existence of a measure or scale where the notion of "reducing" belief towards a point of impossibility exists.

Most of the theories above satisfy this requirement. Probability theory defines a scale on $[0, 1]$ where 0 can be interpreted as impossibility and 1 can be interpreted as certainty. Subjective logic, DST, and possibility theory define multiple measures that enable the expression of more nuanced concepts, such as distinguishing between uncertainty and ignorance, which might be useful for modelling the effect of defeaters on a claim. For instance, in DST, the increase in uncertainty can be modelled as an increase in the level of "disbelief" in a claim, while ignorance about the truth of a claim can be expressed by assigning belief mass to multiple outcomes.

Fuzzy set theory is the only theory that does not readily provide an interpretation for reductions in belief. But this is because fuzzy sets are defined over arbitrary domains that do not necessarily provide interpretations. For instance, a fuzzy set can

be defined over possible heights of human bodies, which is not usually[8] connected to the certainty of claims made in an AC. The above reasoning does not preclude the use of fuzzy sets with a CAM. However, it does show that, if fuzzy set theory is to be used, it should be paired with an additional theory that provides a domain for the fuzzy sets that includes an interpretation of relevant measures of imperfect knowledge.

R3 - Subjectivity

Requirement R3 is focused on mitigating the impact of subjectivity on confidence assessments. The biggest source of subjectivity for confidence assessment is in providing inputs to a method, such as belief valuations or weighting the importance of reasoning steps. There might be many ways for a CAM to mitigate subjectivity in terms of its procedures and application criteria. However, in terms of the underlying theory, two elements contributing to subjectivity are considered. First, the theory should permit the expression of vague or imprecise knowledge. This will avoid situations where users must distill their thinking to a level of precision that is not commensurate with the nature of the subject at hand, compounding the issue of subjectivity. For instance, two experts performing the same assessment might disagree between assigning a belief measure of 0.62 vs. 0.71, but they might more readily agree that their belief is "moderate" as opposed to "low" or "high". As a second mitigation, users should be able to express ignorance (i.e., "I don't know") which might avoid situations where users are required to express an artificially firm belief, adding to the variability of their assessment.

Probability theory on its own does not satisfy this requirement as it requires a user to define probabilities for their belief as singular numbers, possibly with a high degree of precision. However, subjective logic represents opinions as tuples of probability measures which correspond with probability distributions, also permitting the expression of vague knowledge and ignorance [151]. DST can express ignorance, but does not on its own permit the expression of vague knowledge. One of the core values of fuzzy set theory is its ability to express vague knowledge; but it does not have a mechanism for expressing ignorance. Possibility theory's relationship with fuzzy set theory means it can be used to express vague knowledge. Moreover, the

[8]For completeness, one could imagine situations where body height is relevant to an AC's claims, such as for autonomous vehicle perception or in health applications, but the main point still holds, which is that fuzzy set theory does not directly define a domain related to imperfect knowledge.

dual measures of possibility and necessity mean it can also express total ignorance.

R4 - Interpretation

Requirement R4 is focused on the interpretability of outputs produced by a CAM, which is especially important given the role of CAMs as a communication tool. The theory underpinning a CAM should provide support for clearly communicating about whether there is an acceptable level of confidence in an AC.

As with subjectivity, probability theory does not satisfy this requirement on its own because it uses singular numbers that might be expressed with more precision than can be justified for belief assessments. Even if imprecise probability valuations are used (e.g., rounded to the nearest tenth), repeated operations to combine probabilities (e.g., multiplication to compute the probability of a conjunction) result in increasingly fractional probabilities which then result in precise outputs. Other theories, like subjective logic, DST, and possibility theory improve the state of affairs by permitting multi-faceted valuations that capture vague knowledge or ignorance. However, even with interpretation guides, measures expressed on $[0, 1]$ are prone to misinterpretation as frequentist probabilities by non-expert interest holders, due to the prevalence of this type of probability in everyday life. Consider a situation where the results of a confidence assessment are shared with a broader (non-expert) audience: "We are 88% confident that our system will not harm anyone!"[9]. This statement could easily be misinterpreted as 12 out of 100 times the system will harm someone, when in fact 0.88 might correspond to a rather strong level of belief in an AC's claims. Fuzzy set theory (and therefore also possibility theory) could be used to describe CAM outputs linguistically, in a manner that resists misinterpretation.

R5 - Understandability

Requirement R5 is about the understandability of a CAM's mathematics and mechanics for propagating belief through an AC. Importantly, as a prerequisite for trusting a CAM, practitioners must be able to understand its mechanics. The choice of theory for a CAM will have an impact on its understandability in at least two ways: 1) the familiarity of the theory to practitioners, and 2) the complexity of the operators used to manipulate quantities in the theory.

[9]When communicated by experts, this result might be correctly framed as part of a belief scale. However, as in the children's game "telephone", such nuance is easily lost. In short "numbers carry".

Since probability theory is widely taught in post-secondary degree programs, most AC practitioners will be familiar with this theory. Additionally, the operators (addition, multiplication) used by the theory amount to basic arithmetic. Though subjective logic is founded on probability theory, its operations are more complex and are likely unfamiliar to practitioners. While most practitioners will be familiar with basic set theory, fuzzy set theory might be novel; though it is not especially demanding to learn and it benefits from intuitive graphical representations of membership functions. Additionally, operations on fuzzy sets are straightforward extensions of basic set theoretic operations (e.g., union, intersection). While possibilistic logic is likely unfamiliar for most AC practitioners, its operations are straightforward to understand (e.g., min and max).

R6 - Expressivity

Requirement R6 is about the expressivity of a CAM. In particular, the theory's ability to express different types of operations a user might wish to use to determine the belief in a parent claim based on its children. In principle, a CAM can build a wide range of operations on top of a theory, so at this level we consider the core operators that a theory provides "out of the box" and whether they are applicable to reasoning in an AC.

All theories support primitive logical combinators such as "AND" and "OR". Moreover, these theories also provide operators for reasoning, such as basic *modus ponens* to formulate conditional expressions (i.e., "IF x THEN y"). However, ACs might contain reasoning that requires more sophisticated operations to propagate belief. For example, Idmessaoud et al. introduced the "hybrid" operator as part of their DST-based method that balances the extremes of a disjunctive and conjunctive combinator, which can be useful in some situations [50]. As another example, Herd et al. suggest that one of the benefits of subjective logic is the ability to define a wide range of combination operators [154]. However, in both cases, extra machinery was added on top of the theory, and even then available operators might not be sufficient for all reasoning steps in an AC.

Consider Figure 5.6, which contains the first step in the ACC argument fragment used in Chapter 4 (see Figure 4.2 for full example). How should the belief in the child claims, `C6100` and `C6200`, be combined to produce a belief assessment for the parent `C6000`?

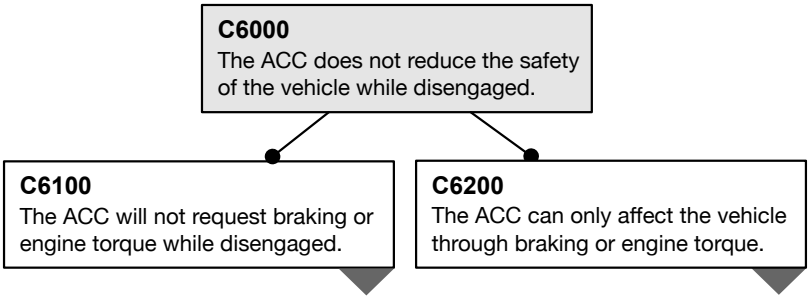


Figure 5.6: Adaptive Cruise Control argument fragment.

Naïvely, the minimum or maximum belief among children could be taken. However, examining the text of the claims more closely suggests a more nuanced propagation rule might be preferred: the belief that the ACC system does not reduce safety while disengaged (`C6000`), depends on the fact that the ACC will not request braking or engine torque while disengaged (`C6100`), but only if there is a sufficient level of belief that there are no other ways the ACC can affect the vehicle (`C6200`). So, the propagation rule should be:

> *If belief in `C6200` is at least [X], then belief in `C6000` should be the same as the belief for `C6100`, otherwise the belief should be the minimum belief among `C6100` and `C6200`.*

As an alternative, suppose directly accepting the minimum belief among `C6100` and `C6200` is acceptable, but that there is some uncertainty about the completeness of the reasoning at this step. Then the propagation rule might be:

> *The belief in `C6000` should be the minimum among `C6100` and `C6200`, up to a maximum of [Y].*

It does not appear as though any theory can directly express either of these propagation rules, at least not without composing multiple operators together. Moreover, existing CAMs do not seem to be able to express these relationships either[10]. This suggests that more flexibility is required in a CAM than can be provided by a handful of pre-defined operators. That is, users should have the option to compose their own belief propagation operations as part of a CAM, similar to how a programming language allows a programmer to compose simple operations into complex programs.

[10]Both Hobbs and Lloyd's BBN method and Idmessaoud's DST method come close, but end up "blending" belief measures together, rather than applying the wanted conditionals or min/max operations.

Summary of Evaluation

The analysis of theories presented above is summarized in Table 5.1. No single theory supports all requirements. However, setting aside requirement R6, a combination of fuzzy set theory and possibility theory is suitable as the basis of a CAM. Possibility theory provides measures for describing belief in claims, including expressing ignorance about a claim's truth, which will be useful for dialectic reasoning and mitigating subjectivity. Fuzzy sets can be used to provide linguistic representations of belief to aid in interpretability of results, and fuzzy set theory (and to an extent possibility theory) is easily understood by practitioners.

Interestingly, no theory (on its own) provides adequate support for expressivity (requirement R6). Moreover, the above analysis suggests that the pre-defined operators offered by existing CAMs do not have enough flexibility to express some types of reasoning steps that might appear in ACs.

Table 5.1: Comparison of theories for imperfect knowledge over selected CAM requirements; strong support: ⊕, some support: ◯, no support: −.

Requirement	Probability Theory	Subjective Logic	Dempster-Shafer Theory	Fuzzy Set Theory	Possibility Theory
R2 (Defeaters)	⊕	⊕	⊕	◯†	⊕
R3 (Subjectivity)	−	⊕	◯	◯	⊕
R4 (Interpretation)	−	◯	◯	⊕	⊕
R5 (Understand.)	⊕	−	−	⊕	◯
R6 (Expressivity)	−	−	−	−	−

† Fuzzy set theory should be paired with another theory to support defeaters.

Towards Representing Imperfect Knowledge in *Certus*

Considering the evaluation of theories presented above, the *Certus* domain specific language defined in subsequent sections of this Chapter will use possibility theory to define a scale for belief that a claim in an AC is true or false. Fuzzy sets will provide an interpretation of vague, but linguistically meaningful, expressions about

belief over this scale. Toward flexibility in expression, *Certus* provides pre-defined operators for common types of reasoning steps, but also allows users to express more complex propagation operators.

5.3 Representing Belief in *Certus*

As discussed in Section 4.3.1, it is preferable to reason over *belief* in the truth of claims in an AC, as opposed to confidence. But, before defining the *Certus* language in full, it is necessary to establish a means of formally describing belief. That is, we must define the quantity that expressions in *Certus* manipulate.

This section builds a theoretical basis for reasoning about belief in claims using vague, but linguistically meaningful, expressions. To start, the truth of a claim is formalized using possibility theory, the result is a joint possibility-necessity scale for representing uncertainty in the truth of a claim. Then the scale is re-interpreted in terms of a more familiar belief scale on the interval $[-1, +1]$. Next, fuzzy sets are used to give a formal interpretation of English language expressions for describing belief. Finally, "canonical" fuzzy membership functions are defined to represent common expressions that can be used to reason about belief in a claim, such as "certain" or "skeptical".

5.3.1 The Possibility and Necessity of a Claim

Let the set of all possible claims in an AC's argument be $\mathcal{C}$. Consider a specific claim $c_i \in \mathcal{C}$. The universe of possible outcomes for the claim is $\Omega_i = \{1_i, 0_i\}$. In one universe the claim is false (0_i) and in the other the claim is true (1_i). A possibility distribution exists for this claim: $\pi_i : \Omega_i \to [0, 1]$. When $\pi_i(\alpha) = 0$ the outcome α is believed to be *impossible*, and when $\pi_i(\alpha) = 1$ outcome α is believed to be *possible*. For example, $\pi_i(1_i) = 0$ means that it is impossible for claim c_i to be true.

Possibility and Necessity Measures

Using the distribution π_i, possibility and necessity measures for the claim c_i are defined over a set of outcomes, $A \subseteq \Omega_i$:

$$\Pi_i(A) = \max_{\alpha \in A} \pi_i(\alpha) \quad \text{and} \quad N_i(A) = 1 - \Pi_i(A^C) \tag{5.6}$$

Since the universe Ω_i of outcomes for c_i is binary and the possibility and necessity measures are duals, given the possibility or necessity of one outcome, the measures for the other outcome can be computed and the underlying possibility distribution can be determined. For example, if we are certain that the outcome "c_i is true" (1_i) will occur, then $N_i(\{1_i\}) = 1$. Since $\{1_i\}^C = \{0_i\}$, then $\Pi_i(\{0_i\}) = 1 - N_i(\{1_i\}) = 1 - 1 = 0$, which means that it is impossible for the outcome "c_i is false" (0_i) to occur[11]. The underlying possibility distribution for this example is $\pi_i = \{0_i \to 0, 1_i \to 1\}$.

Since the possibility and necessity of outcomes 0_i and 1_i are linked, going forward we only consider the truth of a claim (i.e., the "1_i perspective") with the knowledge that we can compute the possibility and necessity for 0_i if wanted. For brevity, we denote the possibility and necessity of a claim as $\Pi_i(1_i) \triangleq \Pi(c_i)$ and $N_i(1_i) \triangleq N(c_i)$.

A Scale for Possibility and Necessity

The possibility and necessity measures provide individual scales on the interval $[0, 1]$. These scales can be placed end-to-end to represent the belief in a claim ranging from impossibility to necessary. This joint possibility-necessity scale is depicted in Figure 5.7. Using the points p, q, r, s, and t annotating the scale, an interpretation of this scale is provided in Table 5.2.

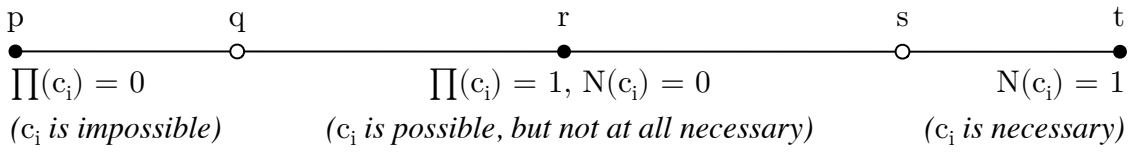


Figure 5.7: Arrangement of possibility and necessity scales to create a joint scale.

Avoiding Inconsistent Knowledge. As discussed above in Section 5.2, it is possible to express an outcome that is both necessary $N(a) > 0$ and also impossible $\Pi(a) = 0$. Such states are inconsistent and must be avoided when reasoning about ACs. However, the joint possibility-necessity scale constructed above avoids situations involving *inconsistent* degrees of possibility and necessity. Mathematically, this is due to the fact that the universe of outcomes for a claim, Ω_i, is strictly binary.

[11]This is somewhat confusing to read due to the double negative, which might lead one to incorrectly interpret this to mean "it is impossible for c_i to be false". Instead, it might be helpful to think of these outcomes (1_i and 0_i) as alternative versions of reality, in one version of reality, c_i is true and in the other c_i is false.

Table 5.2: Interpretation of joint possibility and necessity scales; points refer to annotations on Figure 5.7.

Point	**Possibility** $\Pi(c_i)$	**Necessity** $N(c_i)$	**Interpretation**
p	$\Pi(c_i) = 0$	$N(c_i) = 0$	The claim c_i is impossible and should be rejected as false.
q	$0 < \Pi(c_i) < 1$	$N(c_i) = 0$	The claim c_i is partially possible, but not at all necessary, and is believed to be more likely false than true.
r	$\Pi(c_i) = 1$	$N(c_i) = 0$	The claim c_i is entirely possible and it would not be surprising if it was true, but it is not at all necessary. There is maximum uncertainty about whether c_i is true or false.
s	$\Pi(c_i) = 1$	$0 < N(c_i) < 1$	The claim c_i is partially necessary and is believed to be more likely true than false.
t	$\Pi(c_i) = 1$	$N(c_i) = 1$	The claim c_i is necessary and should be accepted as certainly true.

Defining the possibility or necessity of an outcome implicitly defines the possibility and necessity for the alternative outcome.

A Scale for Belief

The joint possibility-necessity scale shown in Figure 5.7 requires a user to manage two measures for each claim: its possibility and its necessity. In practice, this will be cumbersome and might be non-intuitive for users without an understanding of possibility theory. To better satisfy requirements R4 and R5 from Chapter 4, the function Φ is used to map from a more intuitive belief scale on the interval $[-1, +1]$ onto the joint possibility-necessity scale given above.

First, for a claim c_i we denote the degree of belief as $\phi(c_i)$ or simply ϕ_i with $\phi : \mathcal{C} \rightarrow [-1, +1]$. For now, we assume that belief valuations are assigned by a user.

Next, we represent the possibility and necessity of a claim as an ordered pair $(\Pi(c_i), N(c_i))$ which describes a point on the scale. Then the mapping function is defined $\Phi : [-1, +1] \rightarrow [0, 1] \times [0, 1]$ as follows:

$$\Phi(x) = \begin{cases} (0,0), & \text{if } x = -1 \\ (-x,0), & \text{if } -1 < x < 0 \\ (1,0), & \text{if } x = 0 \\ (1,x), & \text{if } 0 < x < 1 \\ (1,1), & \text{if } x = 1 \end{cases} \tag{5.7}$$

Using semantics from the joint possibility-necessity scale through the mapping function Φ gives an interpretation to the belief values assigned by a user for a claim. For example, $\Phi(\phi_i = 0.25) = (1, 0.25)$ which means that there is a low to moderate degree of belief that c_i is true, per the interpretation in Table 5.2.

It is worth noting that there is nothing inherently special about the interval $[-1, +1]$. Indeed, any non-empty well-ordered set could be used for this purpose and mapped via Φ into possibility and necessity measures. However, the everyday interpretation of the numbers in this range, especially the boundary values -1, 0, and $+1$, is intended to help with interpretation, even for non-experts. A belief value of $\phi_i = -1$ is easily understood as complete rejection of the claim, $\phi_i = 0$ represents a natural point of neutrality or maximum uncertainty, and $\phi_i = +1$ corresponds to certainty in the claim.

5.3.2 Linguistic Reasoning About Claims

So far this section has built up a formalism for reasoning about belief in claims in an AC based on possibility theory. Each valuation of belief for a claim corresponds to a single number in $[-1, 1]$. However, among the challenges with quantitative CAMs discussed in Chapter 3 were issues with interpretation of overly precise confidence (belief) valuations and the subjectivity of assessments, captured by requirements R4 and R5 in Chapter 4. One way to address these challenges is to relax the need for precision when reasoning about belief in an AC. In other words, there is a need to permit *vague* expressions of belief.

In our everyday use of language we reason about degrees of belief using imprecise, but linguistically meaningful, expressions. For example, "I am uncertain if it will rain tomorrow" or "I am fairly sure that we have cat food at home". These expressions are vague in the sense that they do not provide numerical valuations for belief, and yet we are still able to interpret them and make decisions to pack an umbrella or to

purchase another can of cat food[12]. Fortunately, fuzzy sets can be used to represent vague knowledge, providing a bridge between linguistic and numerical representations of belief [183].

In our approach, we construct fuzzy sets over the domain defined by the joint possibility-necessity scale described above. Let $\mathcal{U} = [0,1] \times [0,1]$ be the universe of ordered possibility-necessity pairs. Then a membership function for a fuzzy set $\underset{\sim}{A}$ can be defined $\mu_{\underset{\sim}{A}} : \mathcal{U} \rightarrow [0,1]$ representing the degree of membership of each possibility-necessity pair in the set $\underset{\sim}{A}$.

Suppose that, instead of using the symbol $\underset{\sim}{A}$ as the fuzzy set's name, a word describing a belief level is used. For instance, a set named *high* is depicted in Figure 5.8 such that: pairs with necessity in $[0.6, 0.9]$ have full membership in the set, pairs with necessity in $[0.5, 0.59]$ or $[0.91, 1.0]$ have partial membership in the set, and all other pairs have zero membership. Another set called *skeptical* in Figure 5.8 represents a situation where there is low to moderate belief against the truth of the claim. Using the function Φ from above, the domain of the fuzzy membership functions can be mapped into $\phi \in [-1, 1]$ to represent belief.

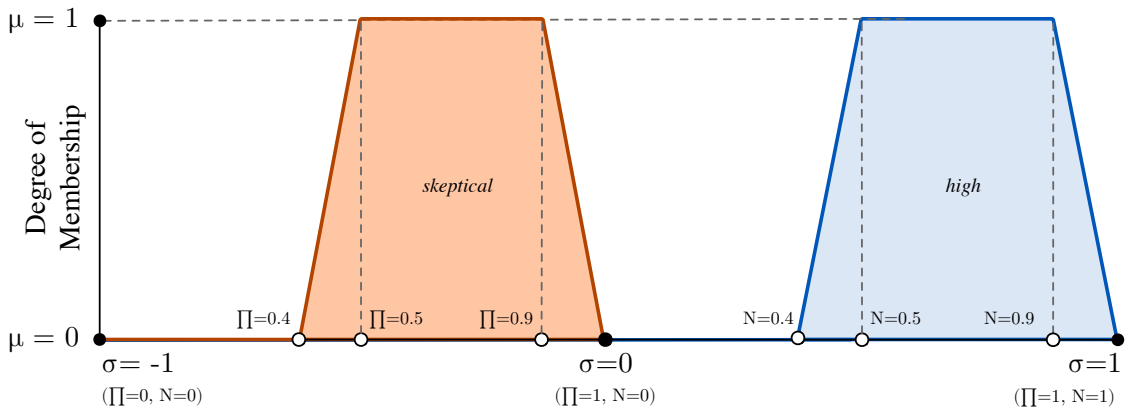
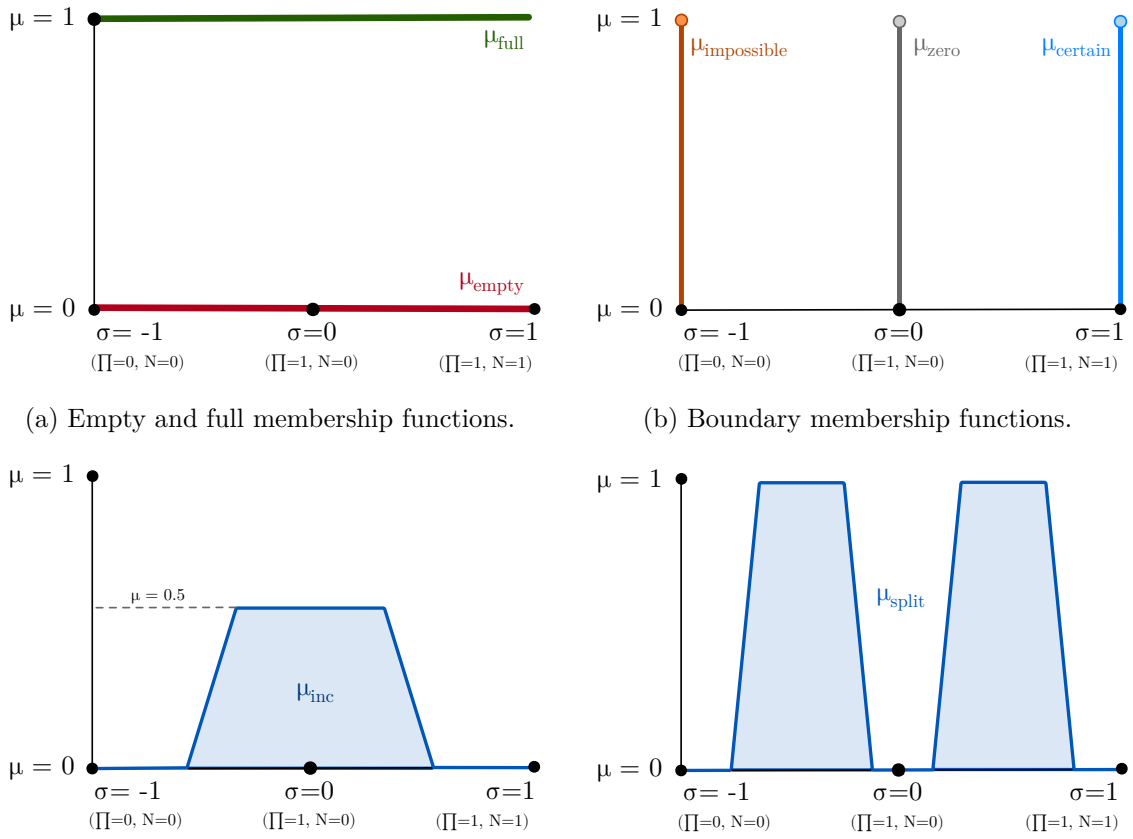


Figure 5.8: Fuzzy membership functions over joint possibility and necessity measures.

Using the same construction as for named sets, the belief in any claim c_i can be represented with an arbitrary fuzzy set over the joint possibility-necessity scale. In doing so, we describe the extent to which each possible belief assignment represents the truth of the claim. Additionally, the same construction can be used to represent belief in the credibility of a defeater, with $\phi_d = +1$ representing the case where the defeater is certainly credible and $\phi_d = -1$ represents the case where the defeater is

[12]When in doubt, always buy more cat food, otherwise your cat will haunt you all night.



(a) Empty and full membership functions.

(b) Boundary membership functions.

(c) Incomplete membership function.

(d) Non-convex membership function.

Figure 5.9: Special cases for fuzzy sets over joint possibility-necessity scale.

not at all credible and should be rejected. We denote the membership function of the arbitrary fuzzy set describing the belief in claim c_i as μ_{c_i} or simply μ_i, and similarly for a defeater D_j as μ_{d_j} or simply μ_j.

5.3.3 Interpretation of Special Membership Functions

Using the above construction, several special cases of membership functions emerge. These are shown in Figure 5.9 and discussed in turn below. In considering these membership functions, additional constraints on the fuzzy membership functions used in *Certus* are identified: they must be both normalized and convex.

The Empty Membership Function. Figure 5.9a shows the "empty" membership $\mu_{empty}(x) = 0$ such that all belief levels have been excluded from the set. For reasoning about claims in an AC, this is an invalid fuzzy set as it does not provide any indication

as to the level of belief in a claim. In fact, as will be discussed below, any non-normal fuzzy set is considered invalid for our approach.

The Full Membership Function. Figure 5.9a also shows the "full" membership $\mu_{full}(x) = 1$ with all belief levels fully included in the set. This corresponds to a maximally vague statement about one's belief in a claim. For example, if Alice asked Bob "Do you think it will rain tomorrow?" and Bob did not reply (i.e., a maximally vague response), we could model Bob's response with μ_{full}. This is different from Bob replying "I don't know", which is expressing a state of maximum *uncertainty* which is depicted as μ_{zero} in Figure 5.9b.

Boundary Membership Functions. Several membership functions in Figure 5.9b correspond to boundary values and are shown as discrete points or "stems". For instance, a claim being impossible is represented by μ_{reject} which is defined as 1 when $\phi = -1$ and 0 otherwise. Similarly, the functions μ_{zero} mean maximum uncertainty and total certainty, respectively. Importantly, while these membership functions express varying levels of (un)certainty in the truth of a claim, they are incredibly precise in their expression, representing belief with a single value.

Incomplete Membership Functions. Figure 5.9c shows a sub-normal membership function, μ_{inc}, that describes some belief values as partially representing belief in a claim, but none that fully describe belief in the claim. As with μ_{empty} defined above, this is an invalid construction for our approach, since it should be possible to define *some* level of belief to every claim, even if that belief is entirely uncertain. More concretely, every fuzzy membership function in *Certus* must be normalized such that $\exists x \in [-1, +1] : \mu_i(x) = 1$.

Non-Convex Membership Functions. It is possible to create non-convex membership functions, for example μ_{split} in Figure 5.9d. This could be taken to mean that one is either in support of, or against, a claim, but is not at all uncertain. While there are surely scenarios in everyday life where an "all or nothing" approach is applicable, in reasoning about ACs this represents a contradiction. Therefore, we require that fuzzy membership functions used in *Certus* are convex.

5.3.4 Canonical Fuzzy Membership Functions

So far we have considered arbitrary fuzzy membership functions as a means of modelling linguistic levels of belief. However, moving forward it is helpful to have a series

of canonical membership functions describing common levels of belief in a claim. Nine functions are defined for this purpose in Table 5.3, with some being depicted in Figure 5.10 for reference. The membership functions are defined such that adjacent functions share a common core with overlapping supports. This means that the intersection of membership functions can be used to create additional sets. For example $\mu_L \cap \mu_H$ defines a triangular function centered on $\phi = 0.5$ that could be used to represent "moderate belief" in the claim.

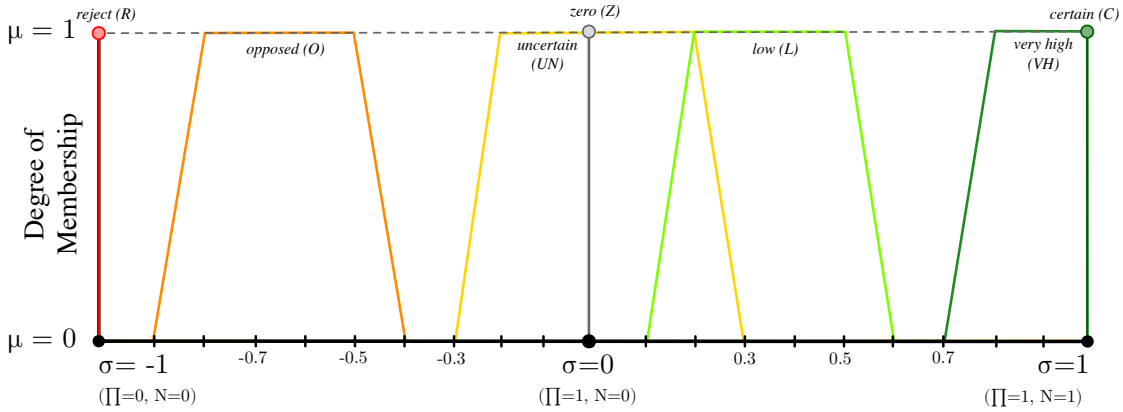


Figure 5.10: Visualization of some canonical fuzzy membership functions.

Table 5.3: Specification of canonical membership functions and their interpretations.

Name	**Abbr.**	**Definition**	**Interpretation**
Certain	C	1 if $\phi = 1$, else 0	The claim is certainly true; it should be accepted.
Very High	VH	$\text{up}(0.7, 0.8)$	Very high belief the claim is true; only slight doubt remains.
High	H	$\text{trap}(0.4, 0.5, 0.8, 0.9)$	High belief that claim is true; some doubt remains.
Low	L	$\text{trap}(0.1, 0.2, 0.5, 0.6)$	Low or weak belief that the claim is true; significant doubt remains.
Uncertain	UN	$\text{trap}(-0.3, -0.2, 0.2, 0.3)$	Uncertain if the claim is true or false; could go either way.
Skeptical	SK	$\text{trap}(-0.6, -0.5, -0.2, -0.1)$	Skeptical that the claim is true; more likely to be false than true.
Opposed	O	$\text{trap}(-0.9, -0.8, -0.5, -0.4)$	The claim is believed to be false; but it is possible it is true.
Very Opposed	VO	$\text{down}(-0.8, -0.7)$	Very strong belief the claim is false; only slight possibility that the claim is true remains.
Reject	R	1 if $\phi = -1$, else 0	The claim is surely false; it should be rejected.

5.4 Definition of the *Certus* Language

Using the formal basis of possibility theory and fuzzy sets established above, this section defines the syntax and semantics for the *Certus* language, using the Lean4 theorem prover and meta-programming environment. Rather than defining every aspect of the language in full detail, the specification and supporting narrative focuses on core elements in the language, such as belief assignment with fuzzy sets and user-defined operators. The full specification is available in Appendix B.

5.4.1 Background on Formal Language Theory

Before defining the *Certus* language, a brief background on formal language theory is provided along with a justification for the denotational approach taken in the remain-

der of the section. The Lean4 meta-programming environment is also introduced to orient the reader to the approach and notation.

Language Syntax

The structure of a language is formalized by a *grammar* which defines permissible sentences or expressions within the language [191]. Formally, a grammar is a 4-tuple (Σ, Δ, P, S) where: Σ is a set of terminal symbols, Δ is a set of non-terminal symbols, P is a set of grammar production rules of the form `X` $\rightarrow$ `Y` with `X` and `Y` being strings composed of symbols from $\Sigma \cup \Delta$, and $S \in \Sigma$ is a start symbol. A *sentence* is considered to be part of the language if it can be generated by successive application of the grammar's production rules in P. The Backus-Naur Form (BNF) of the grammar defines a context-free grammar where the left-hand side of production rules must be a single non-terminal symbol from Σ. The application of a grammar's production rules can be visualized as an *abstract syntax tree* (AST). As an example, consider the language of binary numbers[13]. A grammar expressed in BNF for the binary numbers might have two rules: `B ::= DB | D` and `D ::= 0 | 1`. Applying this grammar to parse the binary string `1101`, produces the AST shown in Figure 5.11.

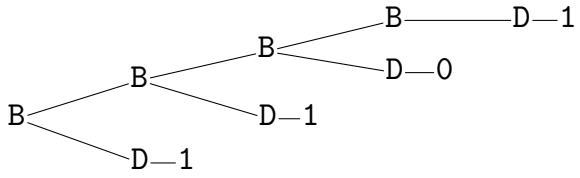


Figure 5.11: Example of the abstract syntax tree for binary string `1101`.

Language Semantics

A formal syntax defines the structure sentences that may be expressed in a language, but it does not ascribe meaning to those sentences. There must be a means of mapping expressions to their meaning, i.e., a semantics. In formal language theory, there are four main approaches for formally defining the semantics of a language, i.e., the meaning of expressions [191] as discussed below.

Attribute Grammars. Knuth introduced *attribute grammars* as a means of computing the values of "attributes" based on context-free grammar production rules

[13]This example is adapted from [192] and also used in [193].

[194]. Each production rule is associated with a formula that manipulates the value of attributes. The value of the attribute(s) produced by applying production rules during language parsing gives the meaning of that expression.

Axiomatic Semantics. The *axiomatic* approach was developed by Hoare as a means defining the effect of a statement in a programming language over a set of abstract variables describing the program's state [195]. Given a state satisfying a pre-condition P, if a statement S in the language terminates, then the post-condition Q is established [191, 196].

Operational Semantics. The *operational* approach defines the meaning of expressions in a language by mapping syntactic structures onto operations performed by a real or abstract machine [191]. For instance, mapping expressions in the C programming language into discrete instructions that change the state of a real device (e.g., a CPU or computer's memory) provide an operational semantics. Turing machines, or other abstract constructs, may also be used for this purpose [196].

Denotational Semantics. Of the approaches for defining a language's semantics, the *denotational* approach is regarded as the most mathematically rigorous [191]. In this approach, the domain of interest, and operations permitted in that domain, is defined by a *semantic algebra*, then *valuation functions* map grammar production rules into elements of the semantic algebra [197, 192].

For example, consider a denotational semantics for the grammar to produce binary numbers given above. The semantic algebra for this language consists of the domain of natural numbers, $\mathbb{N}$, a series of constants $zero, one, two, ... \in \mathbb{N}$, and two operations: **plus** : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and **multiply** : $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ with the usual meanings from conventional arithmetic. The double square brace notation, $[[x]]$, is used to define valuation functions. For the binary number language there are two, β and δ, one for each production rule, defined with prefix notation:

$$
\begin{aligned}
&\beta : \mathtt{B} \to \mathbb{N} \\
&\quad \beta[[\mathtt{BD}]] = (\textbf{plus}\ (\textbf{multiply}\ \beta[[\mathtt{B}]]\ two)\ \delta[[\mathtt{D}]]) \\
&\quad \beta[[\mathtt{D}]] = \delta[[\mathtt{D}]] \\
&\delta : \mathtt{D} \to \mathbb{N} \\
&\quad \delta[[\mathtt{0}]] = zero \\
&\quad \delta[[\mathtt{1}]] = one
\end{aligned}
$$

To define the formal semantics for *Certus*, the *denotational* approach was used, within the Lean4 meta-programming environment as discussed below. In addition to its mathematical rigour, the denotational approach was selected because the mathematical framework of fuzzy sets and possibility theory provides a basis for the required semantic algebra for representing and manipulating belief.

Tool Support for Language Definition

As briefly sketched above for the binary number example, it is possible to write the denotational semantics for a language "by hand", using the language of mathematics. For instance, the lambda calculus can be used to define the valuation functions. However, as the expressions and models become more complex, it is helpful to have a tool to standardize notation and to check that the mathematical expressions are well-formed.

Selecting a Tool. Many software packages and tools exist that could be used to formalize a language's semantics, including the "Z" notation [198], TLA+ [199], Alloy [200], NuSMV [201], the Z3Prover [202], and Lean4 [203]. For this dissertation, the Lean4 interactive theorem prover was chosen because it provides advanced meta-programming capabilities, a rich library of theorems upon which to build (the `mathlib` [204]), can evaluate expressions, and plugs into modern integrated development environments. Additionally, the other notations and tools considered all require the adoption of a state-based view of the domain of interest, except for the Z3Prover. While it might be possible to model fuzzy sets in a state-based tool, it is somewhat at odds with underlying theory which might lead to awkward constructions. The Z3Prover, while quite powerful as a satisfiability solver, does not provide a meta-programming environment.

Meta-programming in Lean4. A language's syntax and semantics can be implemented and evaluated using Lean4's meta-programming capabilities [205, 206]. The process of specifying language in Lean4 can be performed in six steps:

1. **Model Domain** - Create the mathematical objects that describe the domain of interest. In this step Lean4 is used more as a traditional theorem prover than as a meta-programming language. The `mathlib` can be used as a basis for many constructs in this step.

2. **Create Abstract Syntax Model** - Define an abstract model of the language's syntax using Lean4's inductive type system. When an expression in the language is parsed, Lean4 will convert the human-readable syntax into an abstract syntax tree (AST) over these types. For example, we might define two inductive types for representing binary numbers like:

```
inductive Digit
    | zero
    | one

inductive Binary
    | binaryNumber (b : Binary) (d : Digit)
    | binaryDigit (d : Digit)
```

3. **Define Syntax** - Use Lean4's built-in `syntax` operator to define a series of grammar production rules and terminal symbols appear as quoted strings. Precedence among the production rules is optionally defined with a number following the `syntax` keyword. For a grammar to generate binary strings:[14]

```
syntax:40 "0" : digit
syntax:40 "1" : digit
syntax:20 number digit : number
syntax:10 digit : number

syntax "bit" "{" digit "}" : term
syntax "bin" "{" number "}" : term
```

   The final two syntax declarations (`bit` and `bin`) map the production rules from our binary grammar into `term`'s, which are required by Lean4's parser (i.e., everything must derive from `term`).

4. **Map Syntax Terms to AST** - Use Lean4's `macro_rules` command to define a relationship between the parsed syntax terms and an AST defined on the inductive types from Step 2. The mapping to an AST is demonstrated here for completeness, but then omitted for brevity below as it becomes rather repetitive.

```
macro_rules
    | `(bit{0}) => `(Digit.zero)
    | `(bit{1}) => `(Digit.one)
    | `(bin{$d:digit}) => `(Binary.binaryDigit bit{$d})
    | `(bin{$n:number $d:digit})
        => `(Binary.binaryNumber bin{$n} bit{$d})
```

[14] For readability, "0" and "1" are used; however, these are reserved tokens in Lean4, to run this example "0" and "1" must be replaced with "A" and "B" (or some other binary symbol pair).

5. **Define Valuation Functions** - Define regular Lean4 functions over the inductive types of the AST to compute the result according to the semantic algebra defined in Step 1.

```
def evalDigit : Digit → Nat
    | .zero => 0
    | .one  => 1

def evalBinary : Binary → Nat
    | .binaryDigit d => (evalDigit d)
    | .binaryNumber b d => (evalDigit d) + (evalBinary b*2)
```

6. **Evaluate an Expression** - Finally, use Lean4's `#eval` operator to evaluate an expression in our language. For example, `#eval (evalBinary bin{1 1 0})` produces the base-10 value `6`, as wanted[15].

These steps in the meta-programming process have a clear correspondence with the denotational approach for defining a language's syntax and semantics. Step 1 corresponds to defining the language's semantic algebra. Steps 2, 3, and 4 correspond to defining the grammar for the language. Step 5 corresponds defining valuation functions for the grammar's production rules. Finally, while step 6 is not strictly necessary for specifying a denotational semantics, being able to evaluate expressions is useful for checking one's work during development.

5.4.2 Definition of Semantic Algebras

To define the semantics of *Certus*, the first step is to model the domain of interest in Lean4. Definitions for the Booleans ($\mathbb{B}$), natural numbers ($\mathbb{N}$), strings (`String`), and finite sets (`Finset`) are provided by Lean4 via the `mathlib`, and have the usual notations and definitions, for brevity they are not reproduced here. Key aspects of three semantic algebras are defined in brief, one for fuzzy sets, one for assurance arguments, and one for the *Certus* language context. To improve the readability of the specification, only some operations in each domain are provided and extra annotations required to guide Lean4 to complete proofs are omitted[16]. The full specification is available in Appendix B.

[15] We must put white spaces between the bits ("`{1 1 0}`" instead of "`{110}`") because Lean4's tokenizer splits on whitespace. This limitation can be removed by performing additional preprocessing `macro_rules`, but this detracts from our introductory example.

[16] A consequence of omitting the annotations for guiding the prover is that, if the specifications are copied verbatim into Lean4 they will raise many errors.

Semantic Algebra for Fuzzy Sets

For the purpose of formalizing *Certus*, we define fuzzy sets on a finite domain of $n \in \mathbb{N}$ for $0 \leq n \leq 100$, which we refer to as the "unit interval" and denoted as $\mathcal{I}$. The belief scale described in Section 5.3 is defined for $\phi \in [-1, +1]$, which can be linearly mapped $\mathcal{I}$ by taking $\sigma = -1 \rightarrow n = 0$, $\phi = 0 \rightarrow n = 50$, and $\phi = +1 \rightarrow n = 100$. While recognizing that not all theorems from $\mathbb{N}$ readily transpose onto $\mathbb{R}$, this mapping greatly simplifies the construction in Lean4 and means that expressions that are eventually defined can be fully evaluated[17]. The semantics for *Certus* use operators that could be generalized from $\mathcal{I}$ to $\mathbb{R}$ if wanted. The abridged semantic algebra for fuzzy sets on $\mathcal{I}$ is as follows.

Domains. The algebra for Fuzzy Sets has two relevant domains:

- **Unit Interval:** $\mathcal{I} \subset$ `Finset` $= \{n \in \mathbb{N} : 0 \leq n \leq 100\}$ which is accompanied by the usual operations ($>$, $+$, $=$, max, etc.), and an additional operator `n.val` which returns the actual numerical value of an element, `n`, from a `Finset` in Lean4; and
- **Fuzzy Membership Functions:** $\mathcal{F} = \mathcal{I} \rightarrow \mathcal{I}$, i.e., the set of fuzzy set membership functions on $\mathcal{I}$.

Operations. The operations for fuzzy sets are defined as Lean4 functions:

- **Intersect:** The intersection of two fuzzy membership functions.

```
def union (A B : 𝓕): 𝓕 :=
    λ x => 𝓘.min (A x) (B x)
```

- **Invert:** Computes the "mirror image" of a fuzzy set, per Definition 13:

```
def invert(A :  𝓕): 𝓕 :=
  λ x => (A (100 - x))
```

- **Level Mean:** Computes the level mean of a fuzzy set, required for Yager's ranking function.

```
def levelMean(α : 𝓘) (A : 𝓕): ℚ :=
    let X := (𝓘.all).filter (λ x => (A x) ≥ α)
    let count : ℕ := X.card
```

[17] Even though Lean4 has support for $\mathbb{R}$, since it is primarily a theorem prover, it does not have mechanisms for evaluating expressions in $\mathbb{R}$, or more generally over infinite sets. The `mathlib` provides `Finset`, which is intended for use when sets can be reasonably described as finite.

```
let total : ℕ := Finset.sum X (λ x => x)
total / count
```

- **Rank:** Computes Yager's ranking index as in Definition 19.

```
def rank (A : 𝓕): ℚ :=
    (𝓘.all).sum (λ α => levelMean α A)
```

- **Equality (Rank):** Determines if two fuzzy set membership functions are ranked equally by Yager's ranking index.

```
def eq (A B: 𝓕) : Bool
    := decide (rank A = rank B)
```

- **Less Than (Rank):** Determines if one fuzzy set membership function is ranked less than another, according to Yager's ranking index. A similar operation exists for greater than. Combining with Rank Equality above produces less than or equal and greater than or equal.

```
def lt (A B: 𝓕) : Bool
    := decide (rank A < rank B)
```

- **Contains:** Determines if one fuzzy set is a subset of another, per Definition 16.

```
def contains (A B : 𝓕) : Bool :=
    decide (∀ x, A x ≥ B x)
```

- **Normal:** Determines if a membership function is normal, per Definition 9.

```
def isNormal (A B : 𝓕) : Bool :=
    decide (∃x, (A x).val >= UI_UPPER)
```

- **Up:** Defines the upwards ramping fuzzy membership function as shown in Figure 5.5. Similar operations exist for the downward ramp (`down`) trapezoid (`trap`), and triangle (`tri`) membership functions.

```
def rampUp (a b: 𝓘) : 𝓕 :=
    λ x =>
    if x < a.val then 0
    else if x < b.val then
        let width := b.val - a.val
        let m := 100 / width
        let y := (x - a.val) * m
        (min 100 y)
    else 100
```

Canonical Sets. In addition to the above operations on fuzzy sets, we define constant operators corresponding to the canonical fuzzy sets defined in Table 5.3. As discussed above, these are transposed from the belief scale on $[-1, +1]$ into $\mathcal{I}$. Several examples are:

```
def fz_certain : 𝓕 := λ x => if x == 100 then 100 else 0
def fz_vhigh : 𝓕 := λ x => (rampUp 85 90) x
def fz_uncert : 𝓕 := λ x => (trap 40 45 55 60) x
def fz_skep : 𝓕 := λ x => (trap 20 25 40 45) x
def fz_reject : 𝓕 := λ x => if x == 0 then 100 else 0
```

Semantic Algebra for Arguments

Certus is fundamentally about manipulating belief in assurance arguments. This semantic algebra uses the formal definition of an AC argument from Chapter 2. Indeed, complex models of assurance arguments exist, such as the Structured Assurance Case Meta-Model [26]. For clarity in the semantics, a simple model of an Argument is preferred. When implementing these semantics as part of a tool, it might be desirable to extend these definitions to model additional concepts (e.g., additional node types).

Domains. The algebra for an Argument has three relevant domains:

- **Node Types:** The set of node types is $\texttt{NodeType} = \{\texttt{Premise}, \texttt{Defeater}\}$.
- **Node:** Denote the set of possible nodes as `Node`, where each node is a tuple $(\texttt{id}, \texttt{nodeType}, \texttt{children}, \texttt{expr}) \in (\texttt{Ident} \times \texttt{NodeType} \times \mathcal{P}(\texttt{Node}) \times \mathcal{L}_{certus})$ with $\texttt{id} \in \texttt{Ident} \subset \texttt{String}$ uniquely identifying the node, $\texttt{nodeType} \in \texttt{NodeType}$ giving the type of the node, $\texttt{children} \in \mathcal{P}(\texttt{Node})$ containing the direct children of the node, and $\texttt{expr} \in \mathcal{L}_{Certus}$ being a valid expression in the *Certus* language representing the belief in the node. We represent a node in Lean4 as a `structure` as follows:

  ```
  structure Node where
      id : String
      nodeType : NodeType
      children : List NodeId
      expr : Stmt
  ```

- **Argument:** An `Argument` can be modelled as a singleton tuple $(r) \in (\texttt{Node})$ with r being a reference to the top-level (i.e., "root") node that recursively points to each child node. In Lean4, an `Argument` is modelled as a simple `structure` as follows:

```
structure Argument where
    root : Node
```

Operations. There are no special operations for `Argument` or `Node`.

Semantic Algebra for the *Certus* Context

Certus evaluates belief in an argument one step at a time. Each argument step corresponds to a specific *context* which contains the information required to evaluate belief in a parent node based on the previously computed belief levels for the children. The construction presented below is quite simple, with only one domain element and two operations. Later the context will be extended to include other elements, such as user-defined operators, artifacts, and indicators.

Domains. Denote the set of all possible *Certus* contexts as `CertusCtx`. For the moment, we define this as a tuple: $(\texttt{map}, \texttt{root}, \texttt{children}) \in ((\texttt{Ident} \to \mathcal{F}) \times \texttt{Node} \times (\texttt{Ident} \to \texttt{Node}))$ where $\texttt{map} : \texttt{Ident} \to \mathcal{F}$ is a function mapping valid `String` identifiers to fuzzy sets, `root` is the root of the current (sub-)`Argument` being evaluated, and `children` is a function that maps identifiers to `Node`s that are direct children of the `root`. Lean4 provides a built-in parser for identifiers[18], which appears in grammar production rules as `ident`. Identifiers correspond to either argument nodes (e.g., `C1234`) or for named fuzzy sets (e.g., `certain`, `low`). The `CertusCtx` is defined in Lean4 as:

```
structure CertusCtx where
    (map : String → 𝓕)
    (root : Node)
    (children : String → Node)
```

Operations. For the presently simple *Certus* context, there are only three operations which are defined as Lean4 functions:

- **Initialize:** This operation initializes the `CertusCtx` object. Of particular importance is initializing the `map` function with named canonical fuzzy sets.

```
def init : CertusCtx := {
    map := λ s =>
        match s.toLower with
        | "certain" => fz_cert
```

[18] In Lean4 `ident` matches single tokens that contain `_`, `A-Z`, `a-z`, `0-9`, `'` (apostrophe), and unicode characters (for Greek letters).

```
            | "vhigh" => fz_vhigh
            ...
            | "uncert" => fz_uncert
            | "skep" => fz_skep
            ...
            | "reject" => fz_zero
            | _ => sorry

        root := sorry
        children := λ _ => sorry
    }
```

The above definition of `init` occasionally outputs `sorry`, which is a special keyword in Lean4 that signifies an incomplete proof. It could be considered analogous to a "not implemented" exception in an object-oriented programming language. Section 5.4.8 will discuss error handling in *Certus*, including cases where invalid symbols are passed into mapping functions.

- **Set Fuzzy Set:** The `set` operation creates a new `CertusCtx` with an updated `map` function that maps a newly provided fuzzy set to a new name.

```
def set (σ : CertusCtx) (s: String) (fz: ℱ): CertusCtx := {
    map := λ s' => if s = s' then fz else σ.get s',
    root := σ.root
    children := σ.children
}
```

- **Add Child:** The `addChild` operation creates a new `CertusCtx` with an updated `children` mapping function that maps the added node by its identifier.

```
def setNode (σ : CertusCtx) (n: Node): CertusCtx := {
    map := λ x => σ.get x
    root := σ.root
    children := λ n' =>
        if n.id = n' then n else σ.children n'
}
```

5.4.3 Definition of Belief Assignment

The simplest statement in *Certus* is a belief assignment using the `is` operator. For example, the assignment expression `C1 is vhigh` indicates that the belief in the truth of the assertion in node `C1` is *very high*. The left side of this expression identifies a node in the `CertusCtx` (`C1`). The right side is an expression that evaluates to a fuzzy

set. In this case the name of the fuzzy set `vhigh` is given, which evaluates directly to the corresponding fuzzy set in the `CertusCtx`.

Provided the result is a fuzzy set, more complex expressions can be provided on the right side of the `is` operator. For example, `C1 is uncert union skep` creates a new fuzzy set from existing canonical sets or `C1 is up(87, 89)` creates a new upward ramp fuzzy set. Node identifiers also evaluate to their corresponding fuzzy set, if they are defined in the `CertusCtx`. For example, suppose `C1` has a child `C11`, then the expression `C1 is C11` assigns `C1` the same fuzzy set as for `C11`.

The remainder of this sub-section defines the syntax and semantics for basic belief assignments using Lean4's notation.

Abstract Syntax Model for Belief Assignment

To formally define the belief assignment operation in Lean4, two inductive types are required, one for representing expressions yielding fuzzy sets (`FuzzyExpr`) and another for representing a statement in the *Certus* language (`Stmt`). For the moment, the `Stmt` definition is fairly simple, and will be expanded upon later.

```
-- Represents an expression that evaluates to a fuzzy set
inductive FuzzyExpr
    | var (n : String)
    | intersect (e1 e2 : FuzzyExpr)
    | invert (e : FuzzyExpr)
    | min (es : List FuzzyExpr)
    | up (a b : ℐ)

-- Represents a statement in the Certus language
inductive Stmt
    | is (i1: String) (e2: FuzzyExpr)
```

Syntax for Belief Assignment.

Two non-terminal symbols (`stmt` and `fuzz`) are defined, one for each inductive type from above. Then they are used to compose grammar production rules to define the syntax for belief assignment expressions. These rules make use of Lean4's built-in `ident` non-terminal, which matches valid identifier strings for names of fuzzy sets (e.g., `vhigh`) or argument nodes (`C1234`). Finally, the rules for the `min` and `max` expression use Lean4's `sepBy1(sym, delim)` function, which parses one or more instances of `sym` (e.g., `fuzz` non-terminal) separated by a `delim` (e.g., "`,`").

```
syntax fuzz "intersect" fuzz : fuzz
```

```
syntax "up" "(" num "," num ")": fuzz
syntax "invert" "(" num ")": fuzz
syntax "min" "(" sepBy1(fuzz, ",") ")": fuzz
syntax ident "is" fuzz : assign
syntax ident : fuzz
```

Valuation Functions for Belief Assignment.

Two valuation functions are required to process belief assignments, one for evaluating `is` statements and one for evaluating fuzzy expressions.

Valuation Function for the `is` Statement. The `is` statement is evaluated by the `evalIsStmt` valuation function for a given `CertusCtx`. It is a higher-order function that returns a new function producing an updated `CertusCtx`. The new `CertusCtx` has an updated `map` with the identifier, `id`, returning the evaluated fuzzy expression produced by invoking the `evalFuzzy` valuation function.

```
def evalStmtIs (σ:CertusCtx): String → FuzzyExpr → CertusCtx :=
    λ id expr => σ.set id ((evalFuzzy σ) expr)
```

Valuation Function for Fuzzy Expressions. A fuzzy expression is evaluated within a given `CertusCtx` to produce a new fuzzy set by the valuation function `evalFuzzy`. If the expression is simply an identifier, then the `.var` case is triggered, and the corresponding fuzzy set for the identifier is retrieved from the `CertusCtx` via the `.get` operation. Other operations recursively evaluate sub-expressions and then apply the corresponding fuzzy set operation from the semantic algebras defined above. The valuation function is:

```
def evalFuzzy(σ:CertusCtx): FuzzyExpr → 𝓕

    | .var v => σ.get v

    | .intersect f1 f2 =>
        FuzzySet.intersect
            (evalFuzzy σ f1)
            (evalExpr σ f2)

    | .invert f =>
        FuzzySet.invert (evalFuzzy σ f)

    | .up a b =>
        FuzzySet.rampUp a b

    | .min F => match F with
```

```
| [] => sorry
| [f] => (evalFuzzy σ f)
| f :: fs =>
    let r := (evalFuzzy σ (FuzzyExpr.min fs))
    let x := (evalFuzzy σ f)
    if (FuzzySet.lt x r) then x else r
```

5.4.4 Conditional Belief Assignments

The belief assignment operation defined above allows for belief in an argument node to be described in terms of fuzzy sets (e.g., `c1 is vhigh`), combinations of fuzzy sets (e.g., `c1 is uncert intersect skep`), or belief assigned to child nodes (e.g., `c1 is min(c2, c3)`). However, it is sometimes necessary to express more complex conditional relationships over belief in child nodes, like those discussed in Section 5.1. *Certus*' `cases` expression allows a user to perform conditional belief assignments.

A `cases` expression defines one or more Boolean conditions that each map to a fuzzy expression. Cases are matched in order of appearance. If no cases are matched, then an `otherwise` case is used to return a "default" fuzzy expression. The Boolean expression for each case is composed of the usual Boolean operators (e.g., `and` and `or`) connecting fuzzy set comparison operators (e.g., `c2 <= c3`). For an argument with three nodes `c1`, `c2`, and `c3` where `c1` is the parent of `c2` and `c3`, a possible cases expression for assigning belief to `c1` might be:

```
c1 is cases
    c2 is certain and c3 > high -> certain,
    c2 is certain and c3 > uncert -> low,
    otherwise uncert
```

Building on the definitions provided above for basic belief assignment, the remainder of this sub-section defines the syntax and semantics for conditional belief assignments using the `cases` expression.

Abstract Syntax Model for Conditional Belief Assignment

To add conditional belief assignment it is necessary to amend the existing `FuzzyExpr` inductive type to include an abstract syntax model for the `cases` expression. The extended definition represents a `cases` expression as a list of ordered pairs where the first element of each pair is the Boolean expression to be matched and the second element gives the `FuzzyExpr` that is used for belief assignment, if that case is matched. The `default` case is represented as singular `FuzzyExpr`.

```
inductive FuzzyExpr
    ...
    | cases (E: List (BoolExpr × FuzzyExpr)) (default: FuzzyExpr)
```

Next, a new `BoolExpr` abstract syntax model must be defined to represent Boolean expressions. For example, `gt` represents $>$.

```
inductive BoolExpr
    | id (e1 e2 : FuzzyExpr)
    | eq (e1 e2 : FuzzyExpr)
    | gt (e1 e2 : FuzzyExpr)
    ...
    | not (e: BoolExpr)
    | and (e1 e2: BoolExpr)
```

Syntax for Conditional Belief Assignment

An additional non-terminal symbol, `bool`, is added to represent Boolean expressions. The production rules define the usual symbols for comparison (e.g., `>=` means "greater than or equal"). The syntax definition below omits some operators for brevity. There are several important observations for this syntax definition:

- The two operators, `=` and `==`, are different. The symbol `=` is a rank comparison using Yager's fuzzy set ranking index. The symbol `==` is point-wise equality between the two fuzzy set membership functions.
- The `is` operator, in addition to its use as a keyword for belief assignment, can also be used as a comparison operator. When used for comparison, `is` checks containment. For example, `A is high` evaluates to `true` if a A is a subset of the fuzzy set identified by `high`.
- Like `is`, the `contains` operator also checks containment, but in the other direction (e.g., `high contains A`). Though somewhat redundant, `is` and `contains` are both provided so that users can reason linguistically about belief. Users might prefer one operator over another for some situations.

The grammar production rules for conditional belief assignment are as follows:

```
syntax fuzz "==" fuzz : bool
syntax fuzz "=" fuzz : bool
syntax fuzz ">" fuzz : bool
...
syntax fuzz "is" fuzz : bool
```

```
syntax fuzz "contains" fuzz : bool
...
syntax "not" bool : bool
syntax bool "and" bool : bool

syntax "cases" (bool "->" fuzz ",")+ "otherwise" fuzz : fuzz
```

Valuation Functions for Conditional Belief Assignment

Two changes to the valuation functions are required to include conditional belief assignments: the existing valuation function for fuzzy expressions must be amended to evaluate `cases` expressions, and a new valuation function for Boolean expressions must be created.

Amended Valuation Function for Fuzzy Expressions. The `cases` expression is evaluated recursively by the `evalFuzzy` valuation function to produce a fuzzy set. If the Boolean expression for the first case evaluates to `true`, then corresponding fuzzy expression is evaluated and returned. Otherwise, the valuation recurses for the remaining cases, if any. The recursive base case is represented by the `otherwise` case.

```
def evalFuzzy(σ: CertusCtx) : FuzzyExpr → 𝓕
    ...
    | .cases E default =>
        match E with
            | [] => (evalFuzzy σ default)
            | (expr, fz) :: ES =>
                if (evalBool σ expr) then (evalFuzzy σ fz)
                else (evalFuzzy σ (.cases ES d))
```

Valuation Function for Boolean Expressions. Valuation of Boolean expressions is performed by the `evalBool` valuation function within a given `CertusCtx`. The `evalFuzzy` valuation function is used to evaluate any fuzzy expressions that appear in the arguments to comparison operators.

```
def evalBool (σ : CertusCtx) : BoolExpr → Bool
    | .id f1 f2 =>
        FuzzySet.id (evalFuzzy σ f1) (evalExpr σ f2)
    | .eq f1 f2 =>
        FuzzySet.eq (evalFuzzy σ f1) (evalExpr σ f2)
    | .gt f1 f2 =>
        FuzzySet.gt (evalFuzzy σ f1) (evalExpr σ f2)

    | .is f1 f2 =>
        FuzzySet.contains (evalFuzzy σ f2) (evalExpr σ f1)
```

```
| .contains f1 f2 =>
    FuzzySet.contains (evalFuzzy σ f1) (evalExpr σ f2)

...

| .not e
    => !(evalBool σ e1)
| .and e1 e2
    => (evalBool σ e1) ∧ (evalBool σ e2)
```

5.4.5 User-Defined Symbols

In *Certus* users can create new fuzzy sets to describe their belief using operations like `union`, `intersect`, and `up(a, b)`. So far, inline uses of these operators result in an anonymous fuzzy set that is defined "on-the-fly". However, users might wish to provide a name for fuzzy sets they create so it can be re-used. This is accomplished with the `with` expression, which may optionally appear before an assignment expression. For example:

```
with
    med as low intersect high;
    against as skep intersect opposed
end
C1 is med union against
```

A `with` expression contains one or more fuzzy set definitions denoted by the `as` keyword (i.e., `[name] as [definition]`). Definitions are separated by semicolons. The keyword `end` is used to denote the end of the `with` expression.

Definitions provided before an assignment using `with` are mapped into the current `CertusCtx` used to evaluate the assignment expression, but cannot be used outside that assignment scope. It is possible to make global definitions that are available in every scope using the `global` expression, which behaves similarly to `with`, but cannot precede an assignment. For example, if the expression `global med as low intersect high end` is provided, then the fuzzy set `med` can be used in any *Certus* expression. More details about scope management are discussed in Section 5.4.7.

Abstract Syntax Model for User-Defined Symbols

To add user-defined symbols to the abstract syntax model, the `Stmt` inductive type is amended with a new `compose` constructor to model sequential composition of statements.

```
inductive Stmt
    | is (i1: String) (e2: FuzzyExpr)
    | compose (S : List Stmt)
```

Syntax for User-Defined Symbols

Three new non-terminal symbols are introduced in the grammar for user-defined symbols: `definition`, `localBind` (for the `with` expression), and `globalBind` (for the `global` expression).

An outer `certus {...}` term is also added to encapsulate a statement in the language. Within Lean4, any expression identified by this notation is parsed and evaluated as a *Certus* expression.

```
syntax ident "as" fuzz : definition
syntax "with" sepBy1(definition, ";") "end" : localBind
syntax "global" sepBy1(definition, ";") "end" : globalBind
syntax localBind assign : stmt
syntax assign : stmt
```

Valuation Functions for User-Defined Symbols

User-defined symbols are treated as belief assignments in the abstract syntax model. As a result, valuation function defined above for evaluating assignments, `evalIsStmt`, can be entirely re-used. Recall that this valuation function simply updates the `CertusCtx`'s internal `map` to associate a symbol with a fuzzy set. Scope management for local and global definitions will be discussed below in Section 5.4.7.

Beyond belief assignment, the syntax definition above also introduced the notion of composition of multiple belief assignments, and so it is necessary to define a valuation function that can evaluate sequences of assignments. The `certusEval` valuation function given below accepts a statement, possibly including a composition of belief assignments, and modifies the `CertusCtx` to include the resulting symbol definitions.

```
def certusEval (σ : CertusCtx) : Stmt → CertusCtx
    | .is id expr => evalStmtIs σ id expr
    | .compose S => S.foldl (λ σ' s => (certusEval σ' s)) σ
```

5.4.6 User-Defined Operators

Users of *Certus* might want to define their own re-usable operators, similar to subprograms in general purpose programming languages. As an example, consider the

`cases` expression for `c6200` from Figure 5.1 used to invert the belief from defeater `d6220` up to uncertainty:

```
c6200 is cases
    d6220 is certain -> reject,
    d6220 >= vhigh -> vopp,
    d6220 >= high -> opp,
    d6220 >= low -> skep,
    otherwise uncert
```

Suppose a user would like to generalize and re-use this inversion operation elsewhere. The syntax for `with` or `global` expressions defined above can be extended to define operations that accept typed arguments and return a fuzzy set. Using `global`, the `cases` expression is be generalized to:

```
global invert(d:Defeater) as cases
    d is certain -> reject,
    d >= vhigh -> vopp,
    d >= high -> opp,
    d >= low -> skep,
    otherwise uncert
end
```

Then the belief in `c6200` could be specified as: `c6200 is invert(d6220)`.

Abstract Syntax Model for User-Defined Operators

In the example for `c6000` above, the first and only parameter for a user-defined operation is typed as `Defeater`, indicating that a defeater node should be passed as a parameter. For the purpose of defining the language's semantics, there are four possible parameter types defined in the abstract syntax model: `Premise`, `Defeater`, `Fuzzy`, and `Any`. Two of these are already defined as `NodeType`'s in the domain model of an AC argument above. A new type is created to subsume `NodeType` and that also includes two additional types.

```
inductive NodeType
    | Premise
    | Defeater

inductive Passable
    | node: NodeType → Passable
    | Fuzzy
    | Any
```

The `Fuzzy` type permits to any non-node symbol that evaluates to a fuzzy set (e.g., the canonical set `vhigh`). Finally, the `Any` type permits any of the above, provided it

matches a fuzzy set in the `CertusCtx`.

The abstract syntax model for representing a user-defined operation has three elements, a name, a list of named and typed parameters using the `Passable` above, and a `FuzzyExpr` that is evaluated based on the passed parameters.

```
structure UserOp where
    (name: String)
    (args: List (String × Passable))
    (expr: FuzzyExpr)
```

It is also necessary to amend the abstract syntax model for a `FuzzyExpr` to include an invoked operation (e.g., `myOp(c1, c2)`):

```
inductive FuzzyExpr
    ...
    | invokeOp (opName: String) (params: List (String))
```

Syntax for User-Defined Operators

Grammar production rules are added for both defining operations and for invoking them. Definition occurs in the context of `with` or `global` expressions as described above. Invocations occur as part of fuzzy expressions, also defined above.

```
-- Grammar production rules for defining operations
syntax ident ":" opArgType : typedParam
syntax ident "(" typedParam,* ")" "as" fuzz : definition

-- Grammar production rules for invoking operation
syntax ident : invokedArg
syntax ident "(" invokedArg,*")" : fuzz
```

Amended Semantic Algebra for the `CertusCtx`

Before defining valuation functions, it is necessary to amend the semantic algebra for `CertusCtx` defined to include user-defined operators. First, the `CertusCtx` is re-defined as a four-tuple: $(\texttt{map}, \texttt{root}, \texttt{children}, \texttt{ops})$ where the definition of the first three elements is unchanged, and the fourth element is a mapping of identifiers to user-defined operators, i.e., $\texttt{ops} \in \texttt{Ident} \to \texttt{UserOp}$. The definition of `UserOp` provided above is re-used for this purpose. The re-defined `CertusCtx` is:

```
structure CertusCtx where
    (map : String → 𝓕)
    (children: String → Node)
    (root: Node)
    (ops: String → UserOp) -- Added for user-defined operations
```

Second, the operations for `CertusCtx` must be amended. For example:

- **Initialize:** Add the `ops` mapping function to the context initializer:

```
def init : CertusCtx := {
    ... -- Initializers for other elements unchanged
    ops := λ _ => sorry
}
```

- **Check Types:** A new operation is included to perform type checking on the specified parameter types (`spec`) and the actually passed parameters (`act`):

```
def checkOpTypes (spec: Passable) (act: Passable): Bool :=
    match spec, act with
        | .Any, _  => true
        | .Fuzzy, .Fuzzy    => true
        | .node .Premise, .node .Premise => true
        | .node .Defeater, .node .Defeater => true
        | _, _ => false
```

Valuation Functions for User-Defined Operators

Valuation functions are also updated to define and invoke user-defined operations.

Valuation Function for Defining Operations. The function `evalUserOp` is added that includes a new user-defined operator in the `CertusCtx`. The `certusEval` function is amended as well to handle user-defined operations.

```
def evalUserOp (σ : CertusCtx): UserOp → CertusCtx :=
    λ op => σ.setOp op.name op

def certusEval (σ : CertusCtx) : Stmt → CertusCtx
    ...
    | .userOp x => evalUserOp σ x
```

Valuation Function for Invoking Operations. When invoked, user-defined operations produce a fuzzy set, and so are evaluated as one of the alternatives of the existing `evalFuzzy` valuation function. For the `invokeOp` alternative, the function first checks that the provided arguments match, in both arity and type, with the operation's parameter specification. Then it produces a new `CertusCtx` with a new `map` that has the names of the parameters bound to the passed values. The operation's internal expression is then evaluated, and the result is returned. If parameter checking fails, then Lean4's `sorry` is produced; error handling will be discussed in Section 5.4.8.

It is possible to nest operation invocations such that one operation calls another, or even have operations that are self-recursive. A `depth` parameter is used to bound the number of nested invocations. Otherwise, Lean4 will fail to evaluate the expression because it cannot produce a proof that evaluation will terminate.

The relevant parts of the revised `evalFuzzy` valuation function are shown below. For brevity and clarity, some of the more complex expressions are omitted (see ...). The full specification is available in Appendix B.

```
def evalFuzzy(σ : CertusCtx) (depth : ℕ) : FuzzyExpr → 𝓕
    ...
    | .invokeOp opName actuals =>
        let op := σ.getOp opName
        let spec : List String := op.args.map ·(.fst)
        if spec.length = actuals.length then

            -- Check argument types
            let specTypes : List Passable := op.args.map ...
            let actTypes : List Passable := actuals.map ...
            let typesOK : Bool := (checkOpTypes ...)

            -- Create a new context with parameters bound
            let binds : List (String×String) := spec.zip actuals
            let overlay : String → 𝓕 := ...
            let opCtx : CertusCtx := {
                map := overlay,
                ops := σ.ops,
                root := sorry,
                children := sorry
            }

            -- Evaluate and return the operation's expression
            if typesOK ∧ depth > 0
                then (evalFuzzy opCtx (depth - 1) op.expr)
                else sorry
        else sorry
```

5.4.7 Belief Propagation

With the syntax and semantics defined thus far, it is possible to create *Certus* expressions that describe how belief is propagated through a single step in an argument. However, AC arguments almost always have multiple reasoning steps. So, a valuation function is required to propagate belief through many reasoning steps. In *Certus*, belief is propagated from the leaves of the argument "upward" to eventually produce a

belief valuation of the argument's top-level claim.

The following valuation function traverses an `Argument` and returns an updated `CertusCtx` with the resulting belief valuation for the root.

```
partial def evalNode (r: Node) (σ: CertusCtx): CertusCtx :=
    match r.children with
        | [] => (certusEval σ r.expr)
        | C =>
            let σ' := σ
                |> (CertusCtx.setRoot · r)
                |> (C.foldl (λ α c => CertusCtx.addChild α c) · )
                |> (C.foldl (λ α c => (CertusCtx.set α c.id
                                        (CertusCtx.get
                                            (evalNode c σ)
                                            c.id))) · )
            (certusEval σ' r.expr)
```

In the above valuation function, Lean4's `List.foldl` ("fold left") function is similar to a `reduce` operation in other programming languages. It iterates over the list accumulating the results in the accumulation variable α. Multiple operations are chained together using Lean4's threading operator (i.e., `|>`) with the result of the previous operation identified by the $\cdot$ symbol.

Scope Management During Belief Propagation

In *Certus* there are three types of elements that must have their scope managed: nodes in the argument, user-defined symbols, and user-defined operators. The above `evalNode` valuation function shows how each of these scopes are managed via the `CertusCtx`.

Scope for Argument Nodes. Given an argument step $x_0 \multimap x_1, \ldots, x_n$, a *Certus* expression for evaluating belief in x_0 may reference the belief in the children, $x_1, ..., x_n$. However, the expression is not permitted to reference belief in further descendants (e.g., x_1's children, if any). Each *Certus* expression is evaluated within a `CertusCtx` containing a `map` from `Ident` $\rightarrow \mathcal{F}$, including the identifiers for children. The `CertusCtx` also contains a separate mapping function for tracking the current root's children. The `evalNode` valuation function prepares both of these mapping functions by amending the provided context, σ, to produce a new context, σ', that is used for evaluating belief in the parent node. Note that σ' is only used to evaluate belief in the parent, and is not used to evaluate belief in each child. The original σ is used for this purpose, to avoid "scope leakage". Attempting to access argument nodes

that are not in the current context will result in an error, see Section 5.4.8 below for details.

Scope for User-Defined Symbols. The scope of user-defined fuzzy symbols depends on how they are declared, using either `with` (local) or `global`.

- **Local Symbols** - Locally defined symbols may only appear within the *Certus* expression they precede[19]. As shown in valuation functions for symbol definition in Section 5.4.5, symbols definitions are interpreted as assignments, which sets their value in the current `CertusCtx`. As in `evalNode` above, after the belief in the current root node is evaluated, the context containing the local definitions is not re-used.

- **Global Symbols** - Globally defined symbols are accessible in any *Certus* expression across the entire argument. They appear in the base `CertusCtx`, σ passed into the `evalNode` valuation function and are passed through all evaluations in the recursive call to `(evalNode ...)`. To produce a `CertusCtx` with global definitions the user can invoke `evalCertus` on a global expression, and then pass the resulting context into the `evalNode`, like so:

```
def myArgument := ...
def G := certus{global foo as ... end}
def σ := (certusEval CertusCtx.init G)
#eval (evalNode myArgument.root σ)
```

Scope for User-Defined Operators. The accessibility of locally or globally user-defined operators is the same as for user-defined symbols as discussed above. However, operators also have their own internal scope to consider; i.e., the nodes, symbols, or operations that are permitted within the body of an operator. A user-defined operation may access non-node symbols available in the context, such as those previously defined by the user or canonical sets. Other user-defined operations may also be accessed. However, nodes, even those in the current scope, may not be accessed within the body of a user-defined operation, unless their value is passed as a parameter to the operation. The internal scope of an operator is defined by the `overlay` computed immediately prior to its invocation, as shown in Section 5.4.6.

[19] In early work on *Certus* [85], user-defined symbols and operators defined using `with` were accessible in any scope below their definition in an argument's structure. This has since been restricted to only the children of a node. The `global` option was added to provide access to user-defined symbols and operators throughout an argument.

5.4.8 Error Handling

So far, the specification of *Certus*' syntax and semantics has glossed over error handling, often using Lean4's `sorry` keyword as a stand-in for more robust error management in the valuation functions. However, beyond syntax errors, which are handled by Lean4's built-in parser, there are at least three types of semantic errors that might arise during evaluation of *Certus* expressions. Management of each of these errors is discussed in brief below. The complete specifications are not provided in this section since error handling in Lean4 increases the complexity of expressions. The full specification is available in Appendix B.

Lean4 provides several mechanisms for raising and handling errors. Two are used in *Certus*' specification: the `Option T` type and the `Except S T`. Both types are monadic, which means they can be formally modelled as a *monad*, a construct that is useful for encapsulating context and chaining together operations in a functional programming context. Lean4's monadic error handling views errors (and other related states) as part of the context that is passed through chained function calls. Practically, these two error handling mechanisms behave as follows:

- **Option** - The `Option T` data type indicates that the corresponding object might contain some value of type `T`, or it might contain no value. The expressions using the data must handle both the `some` and `none` cases, which is typically done with Lean4's `match` expression. Suppose we have a function `def sqrt (x: Integer): Option Integer := ...` that returns `none` if the requested square root does not exist. Then the calling code might handle the optional type as:

```
match sqrt(-2) with
    | .some x => ... -- do something with sqrt()'s output
    | .none => ... -- handle the erroneous case
```

- **Except** - The `Except S T` data type behaves similarly to `Option T`, but includes an extra type parameter `S` defining the data type for the error case. If the desired error handling is to generate error messages, then `S` is usually `String`. Suppose instead that we have `def sqrt(x: Integer): Except String Integer`. Then the calling code might handle the exception as:

```
match sqrt(-2) with
    | .ok x => ... -- do something with sqrt()'s output
    | .error e => ... -- handle the erroneous case
```

Unmatched Nodes, Symbols, and Operators

Expressions in *Certus* can include references to argument nodes, symbols that resolve to fuzzy sets, or operators. From a purely syntactic perspective, these appear as identifier strings in `Ident` ⊂ `String`. Only when the AST for an expression is being evaluated can well-formed identifiers be checked for validity in the current `CertusCtx` (i.e., do they match with a known object). Error handling for unmatched node, symbol, and operation identifiers happens in two steps.

First, the `CertusCtx`'s structure contains `Option T`'s, rather than simply `T` data types. When looking up the value of an identifier in the `CertusCtx`, `Option.none` is returned if no match is found. For instance, the `CertusCtx`'s `map` for tracking fuzzy sets is of type: `map` : `String` → `Option` $\mathcal{F}$ with the following initializer:

```
def init : CertusCtx := {
    map := λ s =>
        match s.toLower with
            | "certain" => fz_cert
            ...
            | "reject" => fz_reject
            | _ => none -- Return Option.none

    ... -- Other initializers, also using Option T
}
```

Next, the valuation functions using the `CertusCtx` must unpack the `Option T` value returned by the `CertusCtx` and decide how to handle the case where an identifier does not match an object in the context. Valuation functions are typed as `Except String T`, and raise an error with a message. For example, when evaluating a fuzzy expression, if an identifier cannot be matched in the context, then an error is raised with a message.

```
def evalFuzzy(σ : CertusCtx): FuzzyExpr → Except String F
    | .var v => match (σ.get v) with
        | .some x => .ok x
        | .none => .error ("Symbol not found in context: " ++ v)
    ... -- Other types of fuzzy expressions
```

Type Mismatches for User-Defined Operations

The parameters for user-defined operations are typed. It is possible that an operation will be invoked with incompatible data types. In Section 5.4.6, the semantic algebra of the `CertusCtx` was extended to include a `checkOpTypes` operation. This is used by

the `evalFuzzy` valuation function to check that each value passed to the user-defined operation has a compatible data type. If incompatible types are detected, or if the number of passed values does not match the arity of the operation, then the `.error` case is raised with a descriptive message.

Sub-Normal and Non-Convex Fuzzy Sets

By combining some fuzzy operations in *Certus*, it is possible to produce fuzzy membership functions that are either sub-normal or non-convex. For instance, the *Certus* assignment `C1 is vhigh union skep` produces a non-convex fuzzy membership function. Such membership functions are invalid in the *Certus* language for two reasons. First, as discussed in Section 5.3.3, it is not clear how to interpret sub-normal or non-convex cases in terms of belief in an AC. Second, Yager's ranking index which is used for some of *Certus*'s comparison operators requires both normal and convex membership functions.

The `evalFuzzy` valuation function includes a check to determine if the evaluation of a fuzzy expression has produced a sub-normal or non-convex membership function. The `isNormal` and `isConvex` operations from the fuzzy set semantic algebra are used by a helper `checkFuzzy` which takes an `Except String` $\mathcal{F}$ as input, and returns an error if the set is sub-normal or non-convex; otherwise it returns the same fuzzy set unchanged.

```
def checkFuzzy (fz: Except String 𝓕): Except String 𝓕 :=
    match fz with
    | .ok f =>
        if !f.isNormal then .error ("Sub-normal fuzzy set!")
        else if !f.isConvex then .error ("Non-convex fuzzy set!")
        else .ok f
    | .error e => .error e
```

The `checkFuzzy` helper function is used in the main `evalFuzzy` valuation function to check the result of fuzzy operation immediately before returning.

```
def evalFuzzy (σ: CertusCtx) (fz: FuzzyExpr): Except String 𝓕 :=
    let ret := match fz with
        | .var v => ...
        | .intersect f1 f2 => ...
        ...
    (checkFuzzy fz ret)
```

5.4.9 Section Summary

This section provided a specification for the core elements of the *Certus* language using Lean4's meta-programming environment. Key concepts such as (conditional) belief assignment and belief propagation were formalized. The denotational approach for defining the language's semantics was used, which provides a precise interpretation of each expression grounded in fuzzy set theory and possibility theory. Lean4's error handling mechanisms were employed to respond to evaluation errors, such as expressions that produce sub-normal or non-convex fuzzy sets without clear interpretations within the model of belief established in Section 5.3. Subsequent sections in this chapter will extend aspects of the *Certus* language to improve usability and permit reasoning about dynamic assurance cases.

5.5 Macros in *Certus*

The previous section formally defined the syntax and semantics of the *Certus* language. While the language offers a flexible means of specifying how belief propagates through an AC argument, it might be onerous to write detailed specifications for every argument step. Indeed, it is likely that many argument steps require the same belief propagation logic. That is, the "average" argument step can have belief propagation described by a few re-usable expressions.

The *Certus* language permits users to define their own operations (see Section 5.4.6) and re-use them throughout the argument. Though this approach has its merits, namely customizability, it is not a perfect solution. In particular, user-defined operations accept a fixed number of parameters, and when the operator is invoked the user must specify the values that are passed as input. This couples the belief assessment to the *current* structure of the argument and is brittle in the event that the argument changes. For example, suppose a user has defined an operator `myMin(x1:Premise, x2:Premise)` that returns the fuzzy set representing the smallest belief among the two parameters, and then applies it to an argument step $c_0 \multimap c_1, c_2$ with `C0 is myMin(C1,C2)`. If the argument step changes (e.g., c_2 is removed), then the assignment expression for c_0 becomes invalid and must be revised. For a dynamic AC, where the argument structure or evidence might frequently change, this represents a change management challenge.

To address this challenge, and towards satisfying the requirements R9, R10, and

R11, *Certus* provides a macro mechanism that enables context-sensitive expansions of simple keywords into full *Certus* expressions just before the expression is evaluated. Macros are referenced using the `#MACRO` notation. For example, in the example above, the expression `C0 is #MIN` expands to `C0 is min(C1,C2)`, and if a third child node was added, say c_3, the macro expansion becomes `C0 is min(C1,C2,C3)`, without any adjustments by the user.

Macros are not part of the formally defined *Certus* language. Regardless, their expansion into full expressions provides an unambiguous interpretation of the macro in a specific argument context. It is expected that implementations of the *Certus* language apply the desired macro expansions as part of a pre-evaluation processing step via an *expansion procedure*. For example, Section 5.8 describes an implementation of *Certus* in the TypeScript programming language where macros are expanded by dedicated TypeScript functions based on the current argument context.

When implementing macro expansions, towards Requirement R2, it is important to consider the impact of dialectic reasoning on the belief assessment. For non-macro expressions, users can manually describe how defeaters impact belief assessment. However, since macros are intended to automate belief assessment, cases where defeaters are both parents and children in an argument step must be considered. In Section 4.3.2, we described three different interpretations of defeaters: strict, deductive, and eliminative. Depending on the macro, it is possible that a unique expansion procedure is required for each dialectic interpretation.

This section uses the EA notation to define macros, as described in Chapter 2. In particular, premise-typed nodes (mainly claims) may be challenged by defeaters and concurrently supported by sub-premises. Further, defeaters may be rebutted by child claims or supported by child defeaters. There are some differences among how different notations for representing ACs incorporate dialectics. For instance, in GSN the arrows connecting the nodes, in combination with the node type, indicate how the child is to be interpreted [20]. As a result, notation-specific implementations of these macros might differ slightly, though the overall principle remains the same.

The remainder of this section describes expansion procedures for three macros: `MIN`, `MAX`, and `FUSE`. Three versions of each macro expansion are provided, one for each of the three dialectic interpretations. It is imagined[20] that these macros represent common belief propagation operations and are worth providing in implementations of the *Certus* language. They use heuristics that align with reasonable interpretations

of dialectic arguments on the joint possibility-necessity scale defined in Section 5.3.1.

5.5.1 Macros in the Eliminative Interpretation

As discussed in Section 4.3.2, the eliminative interpretation says that belief in a claim in an AC should increase as reasons to doubt the claim are eliminated. In other words, some "positive credit" for resolving defeaters is applied to a claim. Macros adopting the eliminative interpretation view defeaters as "negative premises" where their inverted belief supports the truth of a parent premise. Similarly, for the case where a defeater-typed node is the parent, belief in child premises is inverted and then used to compute belief in the parent. *Certus*' existing `invert` operator is used for this purpose.

The `MIN` and `MAX` Macros in the Eliminative Interpretation

Consider an argument step with a premise-typed parent and a mixture of premise- and defeater-typed children, $c_0 \multimap c_1, ..., c_i, d_j, ...d_n$. In the eliminative interpretation, the `MIN` macro expands as follows:

```
C0 is min(C1, ..., Ci, invert Dj, ..., invert Dn)
```

The `MAX` macro expands similarly, but with `max` rather than `min`. Further, observe that when a premise is supported by only sub-premises, the `MIN` macro is simply the minimum belief among the children. Similarly, for an argument step with a defeater-typed parent, $d_0 \multimap c_1, ..., c_i, d_j, ...d_n$, the expansion is:

```
C0 is min(invert C1, ..., invert Ci, Dj, ..., Dn)
```

Several examples where the `MIN` macro is used to evaluate belief in a parent are shown in Figure 5.12. In example 5.12a, a claim is supported by a certain sub-claim and challenged by a certain defeater. The belief in the defeater is inverted to give rejection (`R`). Then belief in `C0` is selected as the minimum among certainty and rejection.

Example 5.12b is similar, but the supporting claim and challenging defeater are both rejected. According to the eliminative interpretation, the positive credit may be taken for the rejection of d_2, which is inverted to certainty `C`. However, since this is the `MIN` macro, the minimum still evaluates to rejection due to the rejection of c_1.

[20]To be clear, this is a hypothesis based on my personal experience and intuition about AC arguments. I do not have empirical data to make a strong claim about the minimal or necessary set of macros required by users of a CAM. This is a potential avenue for future work.

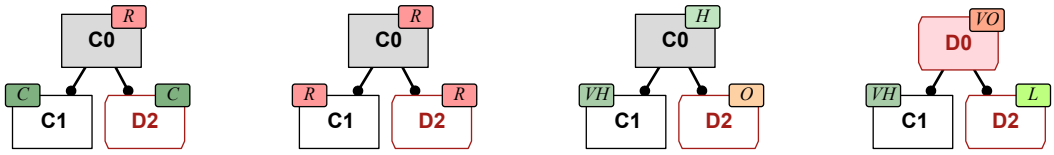


Figure 5.12: Example applications of the eliminative `MIN` macro.

In example 5.12c, a less extreme situation is considered where a parent claim is supported by a child claim with very high (`VH`) belief and also challenged by a defeater that is opposed (`O`). Per the joint possibility-necessity scale, this means that there is some reason to believe the defeater is more likely to be not credible than it is to be credible. Inverting d_2 yields a high (`H`) level of belief. Then the minimum among high and very high belief is high, which is assigned for c_0.

Finally, in example 5.12d a defeater-typed parent is supported by a child defeater with low (`L`) belief in its credibility and rebutted by a child claim with very high (`VH`) belief. Since the parent is a defeater, the belief in the claim is inverted, yielding a very opposed (`VO`) belief against the defeater. Then the minimum among very opposed and low belief is taken.

The `FUSE` Macro in the Eliminative Interpretation

The `FUSE` macro approximates an average among the belief levels of the children in an argument step[21]. In the eliminative interpretation, for a premise-typed parent node, the belief levels of child defeaters are inverted like in the `MIN` and `MAX` macros. Then an "average" among all children is computed.

However, since belief is represented by fuzzy sets, it is not obvious how belief among the child fuzzy sets could be computed. Moreover, the macro must expand to an expression in the *Certus* language, which does not provide an average or mean operator. To address this problem, the canonical fuzzy sets defined by *Certus* are used, which collectively represent a reasonable span of possible belief levels. Each of the nine[22] canonical sets is mapped to an integer in $[-4, +4]$ with -4 representing rejection (*reject*, `R`), 0 representing uncertainty (*uncert*, `UN`), and $+4$ representing certainty (*certain*, `C`). Let $\delta : \mathcal{F} \to \mathbb{Z}$ be a function that maps fuzzy set representations

[21] As will be seen below for other interpretations, this is not true average or mean among the children since some adjustments are made to handle dialectics in different cases. So, I hesitate to call it `MEAN` or `AVG` as this might leave the wrong impression with users. `FUSE` seems an accurate descriptor as it "fuses" together multiple child beliefs according to interpretation-specific heuristics.

of belief into this range with the following definition:

$$\delta(x) = \begin{cases} +4 & x \geq \mathit{certain} \\ +3 & \mathit{vhigh} \leq x < \mathit{certain} \\ +2 & \mathit{high} \leq x < \mathit{vhigh} \\ +1 & \mathit{low} \leq x < \mathit{high} \\ 0 & \mathit{skep} < x < \mathit{low} \\ -1 & \mathit{opp} < x \leq \mathit{skep} \\ -2 & \mathit{vopp} < x \leq \mathit{opp} \\ -3 & \mathit{reject} < x \leq \mathit{vopp} \\ -4 & \text{otherwise} \end{cases} \tag{5.8}$$

Then for an argument step $c_0 \multimap c_1, ..., c_i, d_j, ..., d_n$, the "average" is computed as:

$$\sigma_0 = \left[\frac{\sum_{c \in C} \delta(c) + \sum_{d \in D} \delta(d^I)}{|C| + |D|} \right]_0 \tag{5.9}$$

Where $C = \{c_1, ..., c_i\}$ and $D = \{d_j, ..., d_n\}$ correspond to sets of fuzzy sets assigned to each node in the argument and d^I computes the inverted belief for the defeater[23]. Then the average value can be translated to the equivalent fuzzy set using δ^{-1}. The $[\ldots]_0$ notation denotes truncation (i.e., rounding towards zero), which produces a conservative result that airs towards uncertainty rather than certainty or rejection.

This computation is realized in *Certus* using a `cases` expression by automatically expanding over all possible assignments of canonical sets:

```
with
    idj as invert dj
    ...
    idn as invert dn
end
C0 is cases
    c1 is certain and ... and idj is certain ... -> certain,
    c1 <= vhigh and ... and idj is certain ... -> vhigh,
```

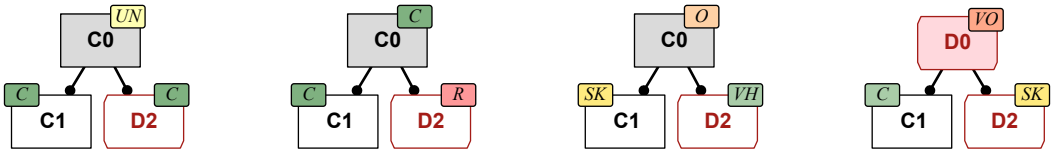
[22]There are actually nine canonical sets defined in *Certus* (see Table 5.3), but *zero* and *uncert* are equivalent under Yager's ranking index and serve the same purpose here, so only *uncert* is used.

[23]Recall Section 5.4.2 and Definition 13 defined fuzzy set inversion, $\underset{\sim}{A}^I$, as computing the mirror image of the set's membership function in the horizontal axis. This is different from the fuzzy set complement, $\underset{\sim}{A}^C$ which is the mirror image in the vertical axis.

```
...
c1 <= uncert and ... and idj is uncert ... -> uncert,
...
c1 <= vopp and ... and idj is reject ... -> vopp,
otherwise reject
```

For a defeater-typed parent the above average computation and macro is the same, except that premise-typed nodes (i.e., claims) are inverted rather than defeaters.

Examples of applying the eliminative `FUSE` macro are shown in Figure 5.13. Example 5.13a shows a scenario where a parent claim is supported by a certain (`C`) sub-claim and challenged by a certain defeater. After inverting the defeater, which gives $\delta(d_2^I) = \delta(certain^I) = -4$, this produces a situation where belief in c_1 "cancels out" the negative belief from the defeater $(+4 + -4)/2 = 0$. As a result, there is uncertain (`UN`) belief in the parent c_0.



(a) (b) (c) (d)

Figure 5.13: Example applications of the eliminative `FUSE` macro.

Example 5.13b illustrates an idealized case, where all defeaters are fully rejected and all supporting claims have certain belief. When it is inverted we have $\delta(d_2^I) = \delta(reject^I) = +4$. Since belief in c_1 is also certain, we have $\lfloor (4 + 4)/2 \rfloor = 4$, which corresponds to certain belief in the parent. This result is consistent with the intuition that, we should be certain in a claim when all of its support is certain, and all reasons to doubt the claim are resolved.

Example 5.13c considers a somewhat messier situation, where there is skepticism about the support for the parent claim and there is very high belief that a challenging defeater is credible. Inverting the defeater gives $\delta(d_2^I) = \delta(vopp) = -3$. Then the average is $\lfloor (-1 + -3)/2 \rfloor = -2$, which corresponds to opposed (`O`) belief in the parent claim.

Finally, example 5.13d shows a situation where a parent defeater's belief is based on the fusion of a certain rebutting claim with a skeptical supporting defeater. Belief in the child claim is inverted, $\delta(c_1^I) = \delta(reject) = -4$. Then the average is computed $\lfloor (-4 + -1)/2 \rfloor = -3$ which corresponds to very opposed (`VO`). This result matches the intuition that a defeater rebutted by a certain claim, with only skeptical support,

should be taken as nearly not credible.

5.5.2 Macros in the Deductive Interpretation

Per Section 4.3.2, in the deductive interpretation, defeaters reduce belief in premise-typed parents, but their resolution does not contribute positively to the belief in the parent. When a defeater is resolved, it returns the belief in the parent to a neutral state. For the deductive macros, it is necessary to define a "deductive heuristic", H, that combines belief from premise- and defeater-typed children according to the deductive interpretation. To build up the intuition for this heuristic, H_2 is first defined for a simple argument step with one child claim and defeater, which is used by the `MIN` and `MAX` macros. Then it is generalized to a belief fusion operator argument steps with multiple premise- or defeater-typed children for the `FUSE` macro.

For the argument step $c_0 \multimap c_1, d_2$, the deductive heuristic should have the following properties:

1. When both c_1 and d_2 are rejected then c_0 should be rejected because there is no reason not to believe c_0 (d_2 is rejected) and the only support for c_0 was rejected.

2. When c_1 is rejected, but d_2 is certain, then c_0 should be rejected, both because the support (c_1) was rejected and because there is a certain reason to doubt c_0.

3. When c_1 has certain belief and d_2 is rejected, then belief in c_0 should be certain because there is certain support for c_0 and no reason to doubt it.

4. When there is certain belief in both c_1 and d_2, then the belief in c_0 should be neutral (i.e., uncertain) since there is both certain support for the claim and a compelling reason to doubt it.

5. When d_2 is not credible (i.e., belief level of uncertain, or lower), then belief in the c_0 should depend only on c_1's belief because there is no credible reason to doubt c_0.

Item 5 above is particularly important for distinguishing the deductive interpretation from the eliminative interpretation above. In the eliminative interpretation, defeaters that are not credible contribute positively to belief in the parent, whereas in the deductive interpretation they have no effect.

Towards the above, the two-node deductive heuristic, $H_2 : (\mathcal{F}_\mathcal{I} \times \mathcal{F}_\mathcal{I}) \to \mathbb{Z}$, for two fuzzy sets (a claim and a defeater) is defined as:

$$H_2(c,d) = \begin{cases} \max(\delta(c) - \delta(d), -4) & d > uncert \\ \delta(c) & d \leq uncert \end{cases} \tag{5.10}$$

Where δ is the mapping function defined above in Section 5.5.1 that produces an integer corresponding to one of *Certus*' canonical fuzzy sets.

It is helpful to visualize H's behaviour in a grid, where the rows correspond to belief in c_1 and the columns correspond to the belief in d_2, as in Figure 5.14. Inspecting this grid confirms that the properties for H identified above are satisfied.

Claim (c1) \ Defeater (d2)	R	VO	O	SK	UN	L	H	VH	C
	-4	-3	-2	-1	0	1	2	3	4
R	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4
VO	-3	-3	-3	-3	-3	-3	-4	-4	-4	-4
O	-2	-2	-2	-2	-2	-2	-3	-4	-4	-4
SK	-1	-1	-1	-1	-1	-1	-2	-3	-4	-4
UN	0	0	0	0	0	0	-1	-2	-3	-4
L	1	1	1	1	1	1	0	-1	-2	-3
H	2	2	2	2	2	2	1	0	-1	-2
VH	3	3	3	3	3	3	2	1	0	-1
C	4	4	4	4	4	4	3	2	1	0

Figure 5.14: Visualization of the H_2 deductive heuristic for argument step $c_0 \multimap c_1, d_2$.

The `MIN` and `MAX` Macros

The deductive `MIN` and `MAX` macros combine the deductive heuristic, H_2, with *Certus*' existing `min` and `max` operators, respectively. For the `MIN` macro, the goal is to compute the most *pessimistic* valuation for the parent, given the belief in the children. Conversely, the goal for the `MAX` macro is to compute the most *optimistic* valuation[24]. To achieve this, for `MIN`, the minimum among the premise-typed children is computed, and the maximum among the defeater type children is computed. Then, the two

values are combined using H_2, to produce the value that is assigned to the parent.

For an argument step like: $c_0 \multimap c_1, c_2, d_3, d_4$, the `MIN` macro expands to the following *Certus* expression:

```
with
    c as min(c1, c2)
    d as max(d3, d4)
end
c0 is cases
    d <= uncert -> c,
    c <= reject and d <= low -> reject,
    c <= reject and d <= high -> reject,
    c <= reject and d <= vhigh -> reject,
    c <= reject and d <= certain -> reject,
    c <= vopp and d <= low -> reject,
    ...
    c <= certain and d <= high -> high,
    c <= certain and d <= vhigh -> low,
    otherwise uncert
```

The macro expansion includes a `cases` expression that implements the H_2 deductive heuristic. Cases in the expression correspond to cells in Figure 5.14, except for the first case in the expression, which covers the entire left-hand side grid.

In cases where there are only premise-typed children, then the expansion is reduced to the minimum belief among the premises. Similarly, in cases where there are only defeater-typed children, the macro is the negation of the maximum defeater score among defeaters with higher than uncertain belief.

When the parent of the argument step is a defeater-typed node, the macro expansion is similar, but the `min` and `max` operations are exchanged and the roles of defeaters and premises are reversed in Equation 5.10 such that defeaters provide support for their parent and premise-typed children rebut the parent.

Consider two example applications of the `MIN` macro in Figure 5.15. In example 5.15a, a claim is supported by two claims with very high (`VH`) and low (`L`) belief, and two defeaters challenge the parent claim with certain (`C`) and high (`H`) credibility. The most pessimistic valuation of this argument is to take the lowest belief among the claims (`L`) and the strongest belief among the defeaters (`C`). Through the deductive heuristic these yield a very opposed belief for the parent claim.

Example 5.15b is similar to the previous example, but considers the case where there is reason to believe that the defeaters d_3 and d_4 are not credible, to varying

[24] The `MIN` and `MAX` macros could arguably be renamed to the `PES` (pessimistic) and `OPT` (optimistic) macros.

(a) (b)

Figure 5.15: Example applications of the deductive `MIN` macro.

extents. The minimum belief among the claims is still low (`L`), but the maximum belief among the defeaters is skeptical (`SK`). Since skeptical is a lower level of belief than uncertain, according to the deductive heuristic, only the claims are considered. The parent belief becomes low (`L`). The intuition is that, if there is uncertainty (or lower) about d_3 and d_4, then we have no reason to believe the defeaters are credible, so they should not count against the parent claim.

Remarks on the Deductive Heuristic. Example 5.15b brings into focus a consequence of property #5 for the deductive heuristic. Though there is skepticism about d_4's credibility, per the joint possibility-necessity scale formulated in Section 5.3, there remains a non-zero possibility that it is a credible defeater. However, this reduction in belief is not taken into account due to H_2's definition. A more "aggressive" deductive heuristic might be preferred where some low level of possibility still counts against the parent claim. However, two challenges emerge when attempting to construct such a heuristic. First, care must be taken to formulate the heuristic so that the operation remains deductive, and does not become eliminative. Second, it is not clear what level of belief "attenuation" should be selected: do *skeptical* defeaters count against the parent, but those that are *opposed* do not? With these problems in mind, the *uncertain* threshold for defeaters was selected for H, and because it represents the point of neutral belief on the possibility-necessity scale. If a more aggressive heuristic is wanted, then the macros in the strict interpretation, described below in Section 5.5.3, can be used instead.

The `FUSE` Macro

The deductive `FUSE` macro applies the generalized deductive heuristic, H, to the current argument step. There are two cases to consider: the parent is a premise-typed node and the parent is a defeater.

Case: Parent is a Premise. Let C and D be sets of premise- and defeater-typed nodes in the argument step, respectively. Then, using the $\delta(x)$ function from Equation 5.8 map fuzzy sets to integers, the generalized heuristic is:

$$H_p(C,D) = \begin{cases} \left[\left(\sum_{c \in C} \delta(c) - \sum_{d \in D} \max\left(0, \delta(d)\right) \right) \cdot |C|^{-1} \right]_0 & |C| > 0 \\ -\sum_{d \in D} \max\left(\delta(d), 0\right) & |C| = 0 \end{cases} \tag{5.11}$$

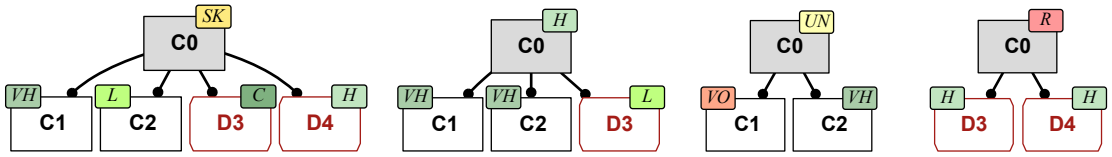
The result from H_p is "clamped" above and below into the range $[-4, +4]$. In the case where there are premise-typed children (i.e., $|C| > 0$), the heuristic computes the average belief among the premise-typed children, after subtracting out the total belief arising from credible defeaters ($\sum_d \max(0, \delta(d))$). The result is truncated (toward zero) to produce an integer result that maps cleanly through δ^{-1} to return the corresponding canonical fuzzy set. In the case where there are no premise-typed children (i.e., $|C| = 0$), the total belief of the credible defeaters is negated and used directly. For the case where there are only premise-type children and no defeaters, H_p becomes the average belief among the premises. Finally, observe that when the argument step contains a single premise and defeater, then H_p becomes H_2.

The expansion procedure for the `FUSE` macro encodes the heuristic H_p as a *Certus* expression. For example, given an argument step $c_0 \multimap c_1, c_2, d_3, d_4$, the expansion looks like:

```
c0 is cases
    c1 is certain and ... and d4 is certain -> uncert,
    c1 is certain and ... and d4 >= vhigh -> uncert,
    c1 is certain and ... and d4 >= high -> low,
    c1 is certain and ... and d4 >= low -> low,
    c1 is certain and ... and d4 >= uncert -> high,
    ...
    c1 >= uncert and ... and d4 >= uncert -> uncert,
    ...
    c1 is reject and ... and d4 is vopp -> reject,
    otherwise reject
```

Several examples of applying `FUSE` are shown in Figure 5.16. Example 5.16a considers a case where there is mixed support for a parent claim from two child claims and a strong challenge from two defeaters resulting in a skeptical parent claim. Example 5.16b shows a case where support from the sub-claims mostly overcomes the doubt from a defeater. Example 5.16c shows a scenario where the heuristic computes

the simple average among child claims, and the average from supporting and opposing child claims cancels out. Finally, example 5.16d shows a situation where only defeaters challenge the parent, and they have enough credibility to reject the parent claim.



(a) (b) (c) (d)

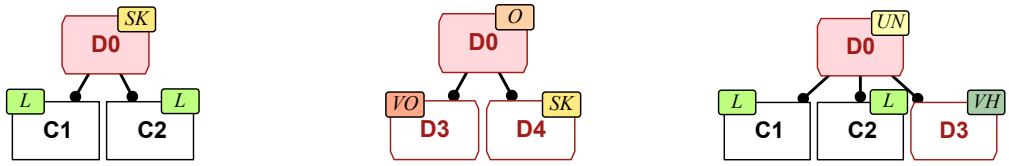
Figure 5.16: Example applications of the deductive `FUSE` macro for a premise-typed parent.

Case: Parent is a Defeater. When the parent is a defeater-typed node, the deductive heuristic is changed in two ways. First, the roles of defeaters and premises are exchanged. Second, the average is taken across all nodes, not just the claim types like in Equation 5.11. The heuristic for a defeater-typed parent is:

$$H_d(C, D) = \left[\left(\sum_{d \in D} \delta(d) - \sum_{c \in C} \max\left(0, \delta(c)\right) \right) \cdot \left(|C| + |D|\right)^{-1} \right]_0 \tag{5.12}$$

As for H_p, the result for H_d is also clamped above and below into the range $[-4, +4]$. Including both $|C|$ and $|D|$ in the denominator of H_d ensures that the heuristic is conservative in the situation where a defeater is rebutted by child premises: each premise must have sufficient strength. Otherwise, many low-belief premises could sum to resolve a defeater, even if no single premise has high belief. The macro expansion of H_d is similar to H_p, but with adjustments to account differences between the respective definitions.

Three example applications of the deductive `FUSE` macro for a defeater-typed parent are shown in Figure 5.17. Example 5.17a shows the parent defeater being rebutted by two child claims, each with low belief, resulting in the skepticism about the defeater's credibility. Example 5.17b shows the average credibility in the parent defeater being computed from two child defeaters, both of which are not credible. Finally, example 5.17c shows two claims rebutting a defeater that is also supported by a defeater with very high credibility, resulting in uncertainty about the credibility of the parent.



(a) (b) (c)

Figure 5.17: Example applications of the deductive `FUSE` macro for a defeater-typed parent.

5.5.3 Macros in the Strict Interpretation

The strict dialectic interpretation says that all defeaters must be entirely resolved (rejected) for a parent claim to be believable. Any unrejected defeater, regardless of the level of belief, fully defeats a parent claim. When the parent is a defeater, more demanding criteria are used: to reject the parent defeater, the child claims must all have certain belief and all supporting child defeaters must be rejected.

The `MIN` and `MAX` Macros

There are two cases to consider for the strict `MIN` and `MAX` macros.

Case: Parent is a Premise. In the case where the parent is a premise-typed node, the `MIN` macro first checks for any unrejected defeaters, if they exist it rejects the parent, otherwise it computes the minimum of the premise-typed children. If there are no premise-typed children and all defeaters are rejected, then `uncertain` is returned. Given an argument step $c_0 \multimap c_1, c_2, d_3, d_4$, the strict `MIN` (and respectively `MAX`) macro expands as follows:

```
c0 is cases
    d3 > reject or d4 > reject -> reject,
    otherwise min(c1, c2)
```

Case: Parent is a Defeater. When the parent is a defeater, the strict `MIN` macro is conservative, and does not immediately reject the parent defeater if child claims exist. To reject the parent, all child premises must be certain and all child defeaters must be rejected. Otherwise, the minimum among the child defeaters is combined with the maximum among inverted claims to give the most conservative valuation of the defeater's belief. Given $d_0 \multimap c_1, c_2, d_3, d_4$, the `MIN` macro expands to:

```
d0 is cases
    c1 is certain
```

```
        and c2 is certain
        and d3 is reject
        and d4 is reject -> reject,
    otherwise min(max(invert c1, invert c2), d3, d4)
```

The FUSE Macro

As with the strict MIN and MAX macros, there are two cases to consider for the strict FUSE macro.

Case: Parent is a Premise. For a premise-typed parent, the FUSE macro first checks for unrejected defeaters, if they exist, then the parent is rejected, otherwise it computes the average of child premises, if they exist. If there are no unrejected defeaters and no child premises, then the parent is taken to be uncertain. More formally, for sets of child defeaters D and premises C, the computation is:

$$S_p(C, D) = \begin{cases} -4 & \exists d \in D, \delta(d) > -4 \\ 0 & |C| = 0 \wedge \forall d \in D, \delta(d) = -4 \\ \left[|C|^{-1} \cdot \sum_{c \in C} \delta(c) \right]_0 & \text{otherwise} \end{cases} \tag{5.13}$$

For an argument step $c_0 \multimap c_1, c_2, d_3$, the strict FUSE macro expands over possible values of c_1 and c_2 as follows:

```
c0 is cases
    d3 > reject -> reject,
    c1 is certain and c2 is certain -> certain,
    c1 is certain and c2 is vhigh -> vhigh,
    ...
    c1 is reject and c2 is vopp -> reject,
    otherwise reject
```

Case: Parent is a Defeater. For a defeater-typed parent, the FUSE macro applies the same reasoning as for the MIN and MAX macros: the parent is only rejected if all child premises are certain and all child defeaters are themselves rejected. Otherwise, the average of child defeater and inverted child claims is computed.

$$S_d(C, D) = \begin{cases} -4 & \forall d \in D, \delta(d) = -4 \wedge \forall c \in C, \delta(c) = 4 \\ \left[(|C| + |D|)^{-1} \cdot \left(\sum_{d \in D} \delta(d) - \sum_{c \in C} \delta(c) \right) \right]_0 & \text{otherwise} \end{cases} \tag{5.14}$$

The results are rounded to $\mathbb{N} \in [-4, 4]$ such that -4 is only produced if all child defeaters are rejected and all premise-typed children are certain. The expansion procedure is similar to the premise-typed parent case. Given an argument step $d_0 \multimap c_1, d_2$, the expansion is:

```
d0 is cases
    c1 is certain and d2 is reject -> reject,
    c1 is certain and d2 is certain -> uncert,
    c1 is certain and d2 is vhigh -> skep,
    c1 is certain and d2 is high -> opp,
    ...
    c1 is reject and d2 is vopp -> uncert,
    c1 is reject and d2 is reject -> uncert,
    otherwise uncert
```

5.5.4 Section Summary

This section has introduced three macros `MIN`, `MAX`, and `FUSE`, and provided definitions for three dialectic interpretations. Macros are useful for encoding re-usable propagation operations that account for local argument context.

Macros are not part of the formal specification for the *Certus* language, and are expanded as part of a pre-evaluation step to produce valid *Certus* expressions. Implementations of the *Certus* language may provide their own macros, using either the definitions provided in this section, or creating their own. Indeed, to accommodate the different dialectic perspectives, the macros defined in this section employ heuristics that encode reasonable propagation logics. However, implementors of the language might express other preferences.

As will be seen in subsequent sections and chapters, macros play an important role in improving the usability of the *Certus* language. Since their expansion is sensitive to the argument context, once employed by the user, they are less brittle under argument change than alternatives, such as user-defined operators or explicit `cases` expressions.

5.6 Assurance Indicators in *Certus*

One of the main objectives of *Certus* is to enable confidence assessment for DACs. Requirement R12 from Chapter 4 requires that *Certus* provide a means to assess confidence (respectively, belief) when performance indicators change their values. As

discussed in Chapter 2, indicators can be used to instrument an AC and provide a live stream of data for continuously validating its claims. It follows that it should be possible to change the level of belief in a node based on the valuation of its indicators.

This section extends the *Certus* language definition provided above to permit expressions that reason about indicators. To further introduce the concept, a example is given showing the intended behaviour of indicators in *Certus*. Then a syntax and semantics is provided as an extension of the specification from Section 5.4.

5.6.1 A First Look at Indicators in *Certus*

Consider a modified version of the adaptive cruise control (ACC) system argument fragment in Figure 5.18. A version of this argument was used to initially introduce *Certus* in Section 5.1 where the belief assessment was "static" in the sense that all belief valuations came from belief assignments provided by users on the leaves of the argument. This version introduces an indicator on `D6220` which turns this argument into a DAC.

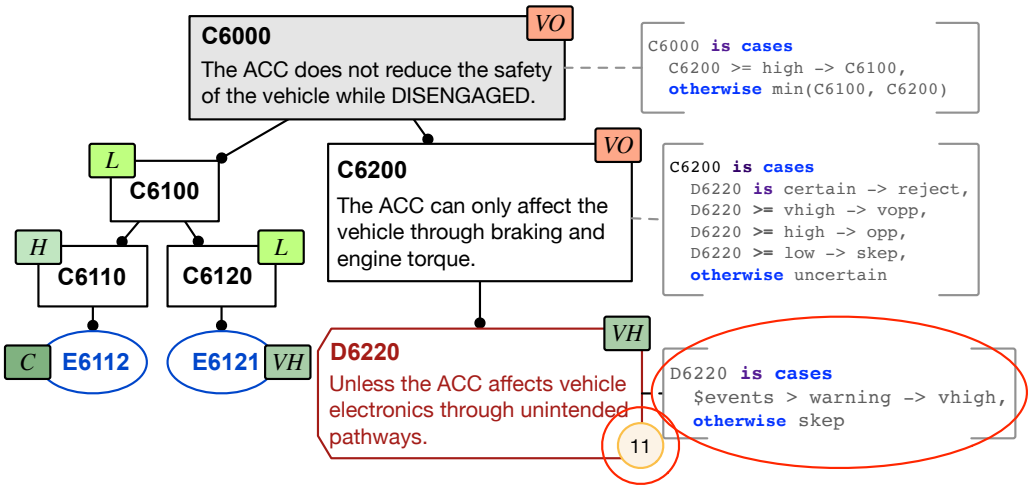


Figure 5.18: Example of an assurance indicator in *Certus*.

Defeater `D6220` is about unanticipated effects the ACC might have on the rest of the vehicle. Suppose that, rather than assigning a static level of belief in the credibility of this defeater, the belief is dynamically evaluated based on an indicator's valuation. Further, suppose the company that developed the ACC has a deployment monitoring program aggregating and analyzing incident reports from the field to identify latent problems with the ACC. One of the metrics monitored by the program

is the number "feature interaction events" per year, across the fleet of monitored vehicles. For brevity, we call this corresponding indicator `$events`. In *Certus* the `$` symbol identifies an indicator[25].

The engineer maintaining the ACC's argument associates the `$events` indicator with `D6220`. In Figure 5.18 the value is `11`, indicating that there were eleven reports of ACC feature interactions from the fleet in the last twelve months. Suppose that for the ACC example, the `$events` valuation function raises a warning if more than ten feature interaction events are reported from the field in the last 30 days. Since the value is eleven the indicator is in the `warning` category.

The *Certus* belief assignment expression for `D6220` includes a `cases` expression where the condition depends on the `$events` indicator. Specifically, if the indicator is in the `warning` category then the belief in the credibility of the defeater is taken as very high (`vhigh`), otherwise there is only skeptical (`skep`) belief in the defeater. Since `$events` is currently in the `warning` category, `D6220` is assigned very high belief which propagates upward to produce an overall belief assessment that shows the top-level claim, `C6000`, is very opposed (`VO`).

Since the indicator is now connected to the argument and embedded in the belief assessment, as the underlying measure changes, the overall belief in the top-level node will be revised. For example, suppose that at the next quarterly safety review the ACC development team finds that the number of `$events` in the last twelve months has dropped to eight. When the value is input into the argument, the belief assessment is revised. The belief in `D6220` becomes `skep`, which propagates upward to produce a top-level valuation of uncertain (`UN`), which is much better than very opposed!

It is worth noting that *Certus* expressions using indicators are not limited to comparisons with categories. Comparisons can be made directly against numerical constants. So, the expression could have been written as: `$events > 10 -> vhigh`, and produced the same result. It is also possible to compare the values of two different indicators, e.g., `$events > $lastYearsEvents`. Finally, indicator comparisons in *Certus* are Boolean conditions, so they can be composed with other Boolean operators to make more sophisticated conditional belief assignments.

[25]In Lean4, the `$` is reserved, so the `!` symbol is used instead for the specification. However, `!` is easily confused as the "not" operator, so it is not a good choice from a usability perspective. I have replaced instances of `!` with `$` below for consistency with the language's implementation, though the raw specification still uses `!`.

5.6.2 Syntax and Semantics for Indicators

To include indicators in the formal specification for *Certus*, the existing semantic algebras, syntax, and valuation functions must be extended, and are discussed in turn below. As with previous semantic definitions, for readability some details are omitted from the Lean4 specification. The complete specification is available in Appendix B.

Semantic Algebra for Indicators

As discussed in Chapter 2, indicators have two fundamental parts, a *metric* giving the current value synthesized from one or more measures, and a *valuation function* that maps to a *category* that often corresponds to an "alarm level" for the indicator (e.g., `ok` or `warning`). There are two domains in the semantic algebra for indicators:

- **Category:** This is the set of possible categories that can be produced by indicator valuation functions. For the purpose of defining these semantics, a small set of categories is considered, but actual implementations of the language might use a wider set of categories, or even permit users to select categories as part of a tool's configuration. Denote the set of all categories as `Category` which is modelled in Lean4 as an inductive type:

```
inductive IndicatorCategory
    | warning
    | ok
```

- **Indicator:** Denote the set of all possible indicators as `Indicator`, where each indicator is a tuple $(\texttt{name}, \texttt{curr}, \texttt{valuation}) \in (\texttt{Ident} \times \mathbb{N} \times \mathbb{N} \to \texttt{Category})$. The current value of an indicator is modelled in $\mathbb{N}$, which is convenient in Lean4; this can be easily generalized to $\mathbb{R}$ for full implementations of the language, which would be required since real-world indicators are often represented with real numbers (e.g., accident rates). In Lean4, this domain is modelled as:

```
structure Indicator where
    (name: String)
    (curr: ℕ)
    (valuation: ℕ → IndicatorCategory)
```

There are no dedicated operations required for `Category` or `Indicator`. However, both the `Node` and `CertusCtx` domains given in Section 5.4 must be modified to include indicators.

The `Node` domain is redefined to $(\texttt{id}, \texttt{nodeType}, \texttt{children}, \underline{\texttt{indicators}}, \texttt{expr}) \in (\texttt{Ident} \times \texttt{NodeType} \times \mathcal{P}(\texttt{Node}) \times \mathcal{P}(\texttt{Indicator}) \times \mathcal{L}_{certus})$, where `indicators` is a list of indicators that are associated with the current `Node`, and the other elements are as before. It is assumed that an external mechanism exists to create associations between nodes and indicators in an AC, perhaps in a tool-based environment, like in [64]. In Lean4 the revised domain is represented as:

```
structure Node where
    id : NodeId
    nodeType : NodeType
    children : List Node
    expr : Stmt
    indicators : List Indicator
```

The `CertusCtx` domain is also redefined to include indicators that are available in the current evaluation context. Only indicators that are associated with the current context root node can be used in the belief assignment expression for the root. The `CertusCtx` is redefined to $(\texttt{map}, \texttt{root}, \texttt{children}, \underline{\texttt{indicators}}) \in ((\texttt{Ident} \rightarrow \mathcal{F}) \times \texttt{Node} \times (\texttt{Ident} \rightarrow \texttt{Node}) \times (\underline{\texttt{Ident} \rightarrow \texttt{Indicator}}))$, where `indicators` is a function that maps from indicator identifiers that appear in belief assignment expressions to the underlying `Indicator`. Operations for the `CertusCtx` are also revised:

- **Initialize:** Include the `indicators` mapping function in the context initializer.

```
def init : CertusCtx := {
    ... -- Initializers for other elements unchanged
    indicators := λ _ => sorry
}
```

- **Set Indicator:** A new operation to add an indicator in the context. This operation updates the `indicators` function, leaving other aspects of `CertusCtx` unchanged.

```
def setIndicator (σ : CertusCtx) (m: Indicator): CertusCtx :=
{
    map := λ x => σ.get x,
    ops := λ x => σ.getOp x,
    children := σ.children,
    root := σ.root
    indicators := λ m' =>
        if m.name = m' then m else σ.indicators m',
}
```

Abstract Syntax Model for indicators

First, the abstract syntax for `Category` is represented by a Lean4 inductive type of the same name, with the two categories described for the domain above:

```
inductive IndicatorCategory
    | warning
    | ok
```

Next, expressions containing indicators must have an abstract syntax model. There are three types of expressions: comparison of an `Indicator` against a `Category`, comparison of an `Indicator`'s value directly against a number using the usual numerical operators like $\geq$, and comparison between the values of two `Indicator`s, also with the usual numerical operators. The abstract syntax model below shows an example for each of these expression types. Where an `IndicatorExpr` has a `String` type argument, the `macro_rules` will provide identifiers for `Indicator` that can be resolved through the `CertusCtx`'s `indicator` function.

```
inductive IndicatorExpr
    -- Compare a Indicator against a category
    | is (n : String) (c: IndicatorCategory)

    -- Compare Indicator's value with a number
    | geqNum (n : String) (val: ℕ)

    -- Compare two Indicator's values.
    | geq (n1 n2 : String)
    ...
```

Finally, all `IndicatorExpr`s are Boolean expressions and so the syntax model for them can be embedded within the existing `BoolExpr` model as one of many potential sub-expressions. This also means that the user can compose multiple `IndicatorExpr`s together using logical connectors, e.g., `cases (!bar >= !foo) or (!bar is ok) -> high, otherwise low`. The revised syntax model for `BoolExpr` is:

```
inductive BoolExpr
    ... -- Previous elements unchanged
    | indicator (e : IndicatorExpr)
```

Syntax for Indicators

In *Certus*, indicators are denoted by the `$` symbol followed by their unique identifier, e.g., `$foo`. The existing grammar is extended to include this notation:

```
syntax "warning" : indicatorCat
```

```
syntax "ok" : indicatorCat
syntax "$" noWs ident : mId
syntax mId "is" indicatorCat : indicatorComp
syntax mId ">=" num : indicatorComp
...
syntax mId ">=" mId : indicatorComp
```

The final rule connects the new `Indicator` production rules with the existing rules for parsing Boolean conditions in `cases` expressions. Lean4's `noWs` operator is used to force matching of '`$name`' (without whitespace), rather than default behaviour which matches whitespace, like '`$ name`'.

Valuation Functions for Indicators

A new valuation function, `evalIndicator`, is added to evaluate indicator expressions. The Lean4 snippet below shows the evaluation for the three types of indicator comparisons: indicator with a category, indicator with a numerical constant, and indicator with another indicator.

```
def evalIndicator (σ : CertusCtx) : IndicatorExpr → Bool
    | .is n c => (σ.indicators n) = (m.valuation m.curr)
    | .geqNum n v => (σ.indicators n) ≥ v
    | .geq n1 n2 => (σ.indicators n1) ≥ (σ.indicators n2)
    ...
```

Then the existing `evalBool` valuation function is amended to include the case where part of a Boolean expression includes an evaluation of an indicator:

```
def evalBool (σ : CertusCtx) : BoolExpr → Bool
    ... -- Other cases unchanged
    | .indicator e => (evalIndicator σ e)
```

Scope Management and Error Handling for indicators

The following describes the inclusion of indicators into *Certus*' evaluation scope and error handling mechanism. See the full specifications in Appendix B for the exact implementation.

Scope Management. Indicators are only accessible within *Certus* expressions if they are associated with the root node of the current argument step. The `CertusCtx` is updated for each argument step to include associated indicators, if any. The `evalNode` valuation function for belief propagation from Section 5.4.7 is updated for this purpose. The change adds an operation that sets the indicators in the `CertusCtx` using

the `CertusCtx.setIndicator` function. Only indicators associated with the current root are set. The resulting `CertusCtx` is used to evaluate the *Certus* expression for the current root node.

Error Handling. It is possible that a user will attempt to use an indicator that does not exist in the current `CertusCtx`. The same error management strategy as for other elements of the `CertusCtx` is used. Namely, an `Option Indicator` type is used to define the `CertusCtx` and `.none` is returned if an unknown indicator is requested. The `evalIndicator` valuation function returns an `Except String Bool` and it returns `.error` if a indicator's name cannot be matched.

5.7 Artifacts in *Certus*

The previous section added indicators to *Certus* to enable dynamic belief assessment of the argument. Change to the evidence supporting an argument is another source of dynamism that lifts a static AC to a DAC, as described by Requirement R11. For instance, in a continuous assurance process the evidence connected to the argument might be updated to reflect the latest verification and validation results for the underlying system. As described in Chapter 2, *artifacts* represent external documents, models, source code, or similar that support the AC's argument. In the EA notation, artifacts are associated to Evidence nodes.

To add artifacts to belief assessment, *Certus* allows users to write Boolean expressions conditioning on whether an artifact is linked to a node in the argument. This means, as artifacts are generated and associated to a DAC, the belief in the argument can be dynamically revised, like in the ENTRUST method [75]. More sophisticated models of evidence and artifact management exist that enable checking of arguments based on detailed criteria [207]. For the time being, *Certus* uses a simple model of artifacts (they are either associated with a node or they are not), with potential extensions for more sophisticated models left as an avenue of future work. Even so, this model is powerful enough to enable dynamic belief assessments as will be shown.

This section extends the *Certus* language definition provided so far to include expressions over artifacts. To introduce and motivate the concept further, a small example is given below showing its use in *Certus*. Then the syntax and semantics are provided as an extension to the specification provided in Section 5.4.

5.7.1 A First Look at Artifacts in *Certus*

Consider the modified adaptive cruise control (ACC) argument in Figure 5.19. Previous versions of this were used to introduce *Certus* and the concept of indicators above. Suppose that the argument is part of a continuous assurance process where a completed AC must be produced for each revision to the system prior to deployment. When the system is changed, the AC's assurance artifacts are disconnected, because they are invalidated by the change. Then, as the artifacts are (re-)created for the changed system, they are associated to the argument's evidence nodes. We would like to model a situation where the belief in an evidence node is conditioned on whether an artifact is associated with it.

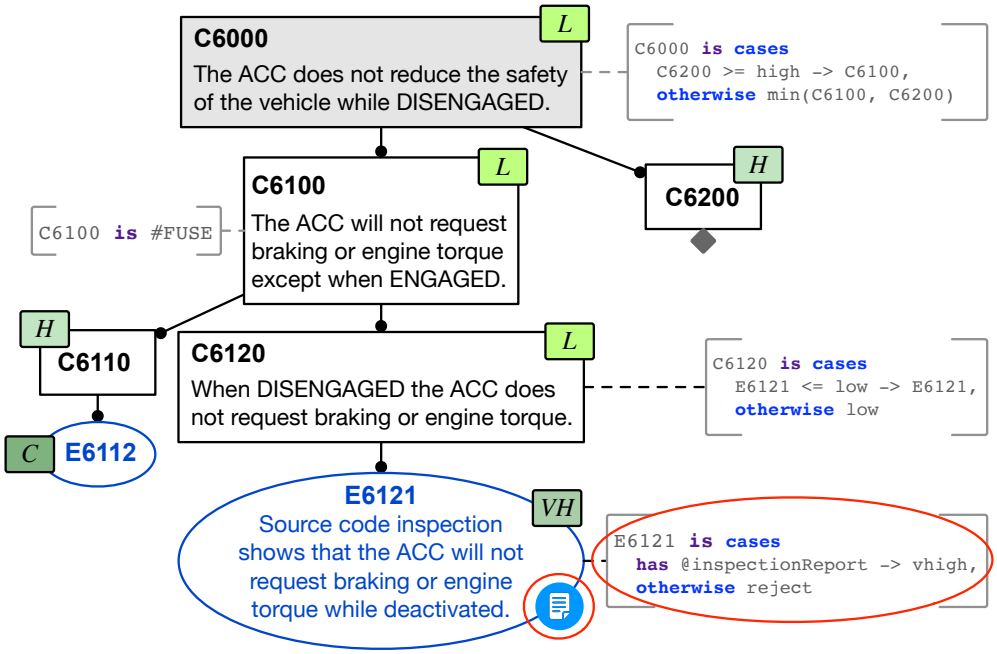


Figure 5.19: Example of an artifact in *Certus*.

In Figure 5.19, one such piece of evidence is a source code inspection, referenced by `E6121`, which supports a claim about the ACC not requesting braking or engine torque when it is in the disengaged state. The *Certus* belief assignment for `E6121` conditions on whether the node has an artifact called `@inspectionReport` associated with it. The `@artifact` notation is used to reference artifacts by name. If the artifact is available, then the belief is taken as very high, otherwise the evidence is rejected as incomplete.

Suppose that initially this piece of evidence is not available, because the previous

source code inspection was invalidated by a change to the ACC's source code. The belief in `E6121` evaluates to `reject`, which propagates upward, resulting in a skeptical level of belief in the top-level claim (since `C6100 is #FUSE` merges the rejection with `high` to give `skep`).

Some time later the ACC development team completes the source code inspection and produces an inspection report. They associate the report with `E6121`. The document icon annotating `E6121` indicates that an artifact is associated with the node. The belief is re-evaluated and since the inspection report is available, the belief in `E6121` is evaluated to `vhigh` and propagated upwards. The *Certus* expression on `C6120` bounds the credit that can be taken for code inspection to at most `low` belief, resulting in a top-level belief of `low` for `C6000`.

5.7.2 Syntax and Semantics for Artifacts

To include artifacts in the formal specification for *Certus*, the existing semantic algebras, syntax, and valuation functions must be updated. As with previous definitions in this chapter, for readability some details of the Lean4 specification are omitted, see Appendix B for the complete specification.

Semantic Algebra for Artifacts

A simple model of an artifact is used: the artifact is either associated with a node or not; so, only a simple domain for artifacts is needed. Let `Artifact` be the set of all possible artifacts, with each artifact represented as a singleton tuple (`name`) $\in$ (`Ident`). This is represented in Lean4 as a structure:

```
structure Artifact where
    (name: String)
```

The `Node` domain is redefined to include a list of `Artifacts` associated with each node. Let each node be a tuple: $(\texttt{id}, \texttt{T}, \texttt{C}, \texttt{M}, \underline{\texttt{artifacts}}, \texttt{expr}) \in (\texttt{Ident} \times \texttt{NodeType} \times \mathcal{P}(\texttt{Node}) \times \mathcal{P}(\texttt{Indicator}) \times \underline{\mathcal{P}(\texttt{Artifact})} \times \mathcal{L}_{certus})$. It is assumed that an external mechanism exists to associate `Artifacts` with `Nodes` in an AC, perhaps as part of a tool. The revised Lean4 specification for `Node` is:

```
structure Node where
    id : NodeId
    nodeType : NodeType
    children : List Node
    expr : Stmt
```

```
indicators : List Indicator
artifacts : List Artifact
```

Finally, the definition of `CertusCtx` must also be updated to account for artifacts that are in the current evaluation context. Let a `CertusCtx` be a tuple $(\texttt{map}, \texttt{root}, \texttt{C}, \texttt{M}, \underline{\texttt{artifacts}}) \in ((\texttt{Ident} \rightarrow \mathcal{F}) \times \texttt{Node} \times (\texttt{Ident} \rightarrow \texttt{Node}) \times (\texttt{Ident} \rightarrow \texttt{Indicator}) \times (\underline{\texttt{Ident} \rightarrow \texttt{Option Artifact}}))$, where `artifacts` is a function that returns `.some` artifact if the provided identifier matches an artifact in the current evaluation context and otherwise returns `.none`. The updated `CertusCtx` in Lean4 is:

```
structure CertusCtx where
    (map : String → Option ℱ)
    (children: String → Option Node)
    (root: Option Node)
    (ops: String → Option UserOp)
    (measures: String → Option Measure)
    (artifacts: String → Option Artifact)
```

The operations for `CertusCtx` are also revised:

- **Initialize:** Include `artifacts` mapping function in the context initializer:

```
def init : CertusCtx := {
    ... -- Initializers for other elements unchanged
    artifacts := λ _ => .none
}
```

- **Set Artifact:** An operation is included to add a new `Artifact` to the current evaluation context:

```
def setArtifact (σ : CertusCtx) (a: Artifact): CertusCtx := {
    map := λ x => σ.get x,
    ops := λ x => σ.getOp x,
    children := σ.children,
    measures := σ.measures,
    root := σ.root
    artifacts := λ a' =>
        if a.name = a' then a else σ.artifacts a',
}
```

Abstract Syntax Model for Artifacts

The abstract syntax model for Artifacts is a single Boolean operator, called `has`, that accepts a `String` corresponding to the artifact's name. This is included as part of the existing `BoolExpr` abstract syntax model, as one of many alternatives:

```
inductive BoolExpr
    ... -- Other alternatives unchanged
    | has (a : String)
```

Syntax for Artifacts

The syntax for referencing artifacts is `@artifact`, which is encoded in the grammar production rules. The `has @artifact` unary operator is also introduced, which is integrated into the existing grammar production rules for Boolean expressions (`bool`).

```
syntax "@" noWs ident : artifactId
syntax "has" artifactId : bool
```

Valuation Functions for Artifacts

To evaluate artifacts the valuation function for Boolean expressions (`evalBool`) is updated to evaluate the `has` operator. The new valuation logic checks if the named `Artifact` exists in the current `CertusCtx`, if it does, then `true` is returned, otherwise `false`.

```
def evalBool (σ : CertusCtx) : BoolExpr → Except String Bool
    ... -- Other alternatives for BoolExpr are unchanged
    | .has a =>
        match (σ.artifacts a) with
            | .some _ => .ok true
            | .none => .ok false
```

Scope Management for Artifacts

This evaluation approach for artifacts does not require any special error handling since the case of an unmatched artifact resolves to `false`. However, as the belief propagation function, `evalNode`, is updated to include artifacts associated with the current evaluation root in the `CertusCtx`. This is achieved via the `CertusCtx.setArtifact` operator from the semantic algebra.

5.8 Implementing *Certus*

So far this chapter has described the mathematical basis and formal specification for the *Certus* language in the Lean4 meta-programming and theorem proving environment. Though it is possible to evaluate *Certus* expressions in Lean4's theorem prover,

it is not well suited for use by end-users. For instance, the specification makes several simplifying assumptions such as the domain of the underlying fuzzy membership functions. Moreover, AC development often occurs in dedicated development tools, so *Certus* should be available in a format that readily integrates with those tools.

This section briefly describes the implementation of the *Certus* language as an open-source TypeScript library, called `certus-ts`. The library is then integrated into an existing commercial AC tool, *Socrates*. Implementing *Certus* as a library, and incorporating it into a commercial tool required addressing a number of practical aspects such as user interface design. This served as a means of preliminary validation of the language, and provided valuable feedback which was used to improve the language's specification.

5.8.1 The `certus-ts` Library

The `certus-ts` provides an implementation of the *Certus* language specification in TypeScript, a popular extension of the widely used JavaScript general-purpose programming language[26]. TypeScript was chosen as an implementation target because it is easily integrated into web and desktop applications, which will aid in the adoption of *Certus*. Additionally, TypeScript provides a layer of data type management on top of JavaScript's type system which helps to avoid software defects arising from type errors. The remainder of this sub-section discusses aspects related to implementing `certus-ts`.

The `vaguely` Library for Fuzzy Sets

Unsurprisingly, TypeScript does not provide native data types for fuzzy sets, and prior to this project, no suitable TypeScript library existed. So, it was necessary to also implement the machinery for creating and manipulating fuzzy sets. This was undertaken as a separate open-source library called `vaguely`[27] which became a dependency for `certus-ts`.

The `vaguely` library contains two core concepts: a `Domain` and a `FuzzySet`. In short, `FuzzySet` membership functions assign membership degrees over a `Domain`. The library provides several common `Domain`s, such as the `UnitInterval` ($[0, 1]$), or customized domains can be specified. Of note is the ability to create both enumerated

[26]The library is available via the Node Package Manager: `www.npmjs.com/package/certus-ts`.
[27]Available via Node Package Manager: `https://www.npmjs.com/package/vaguely`.

domains or domains over intervals in $\mathbb{R}$. The package also has functions for creating and manipulating `FuzzySet`s, including operators such as intersection, union, and complement. For example, the following TypeScript snippet creates two `trap`ezoid membership functions on $[-1, 1] \in \mathbb{R}$ and then outputs their intersection.

```
const I = new NumericalDomain("Belief", -1, 1, 0.01);
const f1: FuzzySet<number> = trap(I, 0.1, 0.2, 0.3, 0.4);
const f2: FuzzySet<number> = trap(I, 0.2, 0.3, 0.4, 0.5);
console.log(f1.intersect(f2));
```

In the formal specification for *Certus* the domain $[0, 100] \in \mathbb{N}$ was used to reduce the number of proof obligations generated by Lean4's theorem prover. However, as seen from above, the `vaguely` library permits a wider range of domains, including the interval $[-1, 1]$ which is used for modelling belief on the joint possibility-necessity scale in Section 5.3.

Parsing *Certus* Expressions with `ANTLR`

The `certus-ts` library uses `ANTLR`4 (ANother Tool for Language Recognition, version 4) to parse *Certus* expressions provided by users [208]. `ANTLR` itself is not a parser, it is a tool for generating parsers in a target programming language. For `certus-ts`, `ANTLR` was used to generate a parser for *Certus* written in the TypeScript programming language. The main input to `ANTLR` is a grammar file describing the language's syntax. A sample of the `ANTLR` grammar is shown below for common Boolean operators.

```
// Boolean expressions (e.g., (foo > 0.1) and (bar > 0.3))
booleanExpr
    : '(' booleanExpr ')'
    | booleanExpr AND booleanExpr
    | booleanExpr OR booleanExpr
    | fuzzyExpr EQ fuzzyExpr
    | fuzzyExpr LT fuzzyExpr
    ...
    ;

// Fuzzy expressions (e.g., foo intersect bar)
fuzzyExpr
    : fuzzyExpr INTERSECT fuzzyExpr
    | ...
    ;

// Operator definitions
AND : '&' | 'and' | 'AND' ;
OR :  '|' | 'or' | 'OR' ;
EQ : '=' | 'eq' | 'EQ' ;
LT : '<' | 'lt' | 'LT' ;
...
```

The ANTLR grammar sample reveals another difference between the Lean4 specification and the implementation by certus-ts. Specifically, there is a wider range of symbols for each operator in certus-ts. For instance, in Lean4 the conjunction (logical AND) operator was represented by and, but in certus-ts several symbols are available: &, and, and AND. Implementing all of these in Lean4 would have made for a lengthy language specification.

Another syntactic difference between the Lean4 specification and the certus-ts implementation is the format of identifiers for indicators and artifacts. In the Lean4 syntax identifiers are not allowed to contain whitespace. However, this is somewhat impractical as both of these elements often have names that naturally contain spaces, e.g., "Number of Accidents" as an indicator name. So certus-ts provides an option for users to surround names with quotations.

```
measureId :
      | '!' '"' (ID WS?)+ '"'
      | '!' ID ;
```

Evaluating *Certus* Expressions

The ANTLR tool uses the grammar to generate a *lexer* and a *parser*. The lexer tokenizes raw input from a user to produce a sequence of tokens that are then used by the parser to construct a parse tree (i.e., an AST). ANTLR also provides classes to operate on the AST. To evaluate an expression, certus-ts uses a *visitor* to traverse the AST. Evaluation is performed within a CertusCtx, and produces and updated context object with the result of the evaluation. In the case of a belief assignment expression, the updated context will have the updated valuation assigned to the node in the CertusCtx. If global is used to define symbols or operators, then the CertusCtx is updated to contain those instead.

Macros. All macros described above in Section 5.5 are included in certus-ts, including variants for different dialectic interpretations (e.g., #FUSE_STRICT). Macros are expanded in a pre-processing step, before the expression is provided to ANTLR's parser. certus-ts implements expansion procedures that are consistent with the macro specifications given above.

Error Management. The lexer and parser generated by ANTLR report syntax errors encountered during parsing. The errors are stored in the CertusCtx and returned to the calling code when parsing is complete. Semantic error checking, such as checking for sub-normal or non-convex fuzzy membership functions, is performed by the visitor

as it evaluates the expression. If semantic errors are detected, then they are also stored in the `CertusCtx` and then returned to the calling code.

Belief Propagation

As described above, the main function of the `certus-ts` library is to evaluate expressions within the provided context, and then to produce an updated context. Each expression in *Certus* corresponds to a single step in an argument. As in the formal specification above, the `CertusCtx` in `certus-ts` contains a root node for the current argument step and the child nodes of the root. The `certus-ts` library does not provide a belief propagation algorithm. The calling software is expected to implement its own argument graph traversal, passing an updated `CertusCtx` for each expression.

Some might regard this as a limitation of `certus-ts` ("why not implement the propagation logic as well!?"). In fact, it arises as a practical constraint for software integration. `certus-ts` is intended to be integrated into existing AC tooling, which will almost certainly have existing data models and highly tuned argument traversal algorithms. The designers of those tools are best positioned to implement an appropriate traversal algorithm for their data model(s). Otherwise, they must write custom "glue" code to translate their data model into *Certus*', which might be non-trivial if the data models are significantly different. From an integration perspective, providing an interface for evaluating *Certus* expressions (not whole arguments) is easier. Moreover, the belief propagation logic specified in Section 5.4.7 is simple enough (it is effectively a depth-first tree traversal) that it can be implemented easily.

Optimizing Macro Evaluation in `certus-ts`

The specification above describes macros being expanded into `cases` expressions prior to evaluation. This is done to aid understandability: a user can inspect the expanded macro to understand how belief is propagated. The expansion is sensitive to the context of the current argument step. The number of cases in the expression grows exponentially with the number of children. For example, for n children and 9 canonical sets, the number of cases for the `#FUSE` macro is 9^n. In practice, this results in large expressions that take a long time to parse and evaluate. For large arguments with many reasoning steps and macros, it takes a long time to compute the total belief in

the argument[28].

The implementation of *Certus* in the `certus-ts` library addresses this limitation in two parts. First, macros are evaluated directly using formulae defined in Section 5.5 above, which significantly improves performance. Second, the interface to the `certus-ts` library exposes an `expand(...)` function that produces the fully expanded macro, if desired. Users may expand and inspect macros to understand their behaviour in a specific argument context. This two-part approach addresses the need for understandability while also maintaining acceptable evaluation performance.

5.8.2 Incorporating *Certus* in *Socrates*

Socrates is a commercial AC development platform developed by Critical Systems Labs Inc., a Vancouver-based engineering firm specializing in assurance for critical systems[29]. The tool is used by a number of organizations that develop or maintain ACs for critical systems in several industries. For instance, it was used to develop an AC for the Large Hadron Collider's beam dumping system (i.e., "shutdown system"), which serves as one of the few publicly available at-scale case studies in the field [174, 91]. Also, *Socrates* already implements other CAMs, including Idmessaoud et al.'s DST method [50] and Hobbs and Lloyd's BBN method [48], extended with dialectic reasoning as described in Chapter 4. Its status as a mature commercial solution for ACs along with its existing infrastructure for confidence assessment (e.g., graphical user interface (GUI) elements), makes it a compelling platform for creating a prototype integration with *Certus*.

Incorporating a prototype of *Certus* into *Socrates* entailed a number of design decisions related to user experience (e.g., workflows, user interfaces, etc.). Importantly, although *Certus* is defined as a textual language, it is not necessary for end-users to interact with it in exclusively this manner. Rather, *Certus* can be viewed as the back-end or "engine" driving a predominantly graphical front-end. In this front-end, the most common operations are exposed via the GUI, with an option for users to author more sophisticated expressions as needed. The following narrative and screenshots walk through the main workflows for performing belief assessment using *Certus* in *Socrates*. The adaptive cruise control (ACC) argument is used as a running example.

[28] Here a "long time" is measured on the order of 10s of seconds. Depending on tooling setup, this might be tolerable. However, faster computations mean that *Certus* can be used across a wider range of user workflows, such as a GUI.

[29] `https://criticalsystemslabs.com/socrates-assurance/`

It is worth noting that *Socrates* is a functionally complete tool that provides a wide range of features for modelling and analyzing ACs. *Socrates* can be used without *Certus*. Similarly, though a prototype interface for *Certus* has been added to *Socrates*, the core elements of *Certus* remain an independent open-source library that can be used independently of any tool.

Visualizing Belief Assessment Results

In *Socrates*, users interact with AC arguments primarily via diagrams in the style of GSN, EA, or CAE. The results of confidence assessments are rendered as annotations in the top-left of each node, as shown in Figure 5.20 for an EA-style argument. Colour coding is used to signal the degree of belief. Annotations with shading in the green-yellow-red spectrum indicate an assigned or computed belief level, with the *Certus* canonical fuzzy sets used to indicate belief levels. Grey shading indicates that a default assignment was provided by the tool, as is the case for `E6112` in Figure 5.20.

Users interact with the belief assessment feature in *Socrates* by double-clicking on the belief annotations. For instance, double clicking on the annotation on `E6112` will open a window for the user to provide a belief assignment.

In addition to the *Certus* annotations, artifact associations are shown as blue icons on the right side of nodes (see `E6112` and `E6121`). Linked indicators appear in the bottom left of nodes (see `D6220`).

Belief Assignments

Once the user has selected a node, there are several ways to provide *Certus* belief assignments in *Socrates*. For leaf nodes, users are presented with a dropdown list of potential belief assignments, corresponding to *Certus*' canonical fuzzy sets, as shown in Figure 5.21. Alternatively, they may author a custom belief assignment expression by selecting "Custom" from the list.

If the user opts for a custom belief assignment expression, then a text editor is revealed as shown in Figure 5.22. The tool also populates a banner of available in-scope context elements, including child nodes, indicators, artifacts, and user-defined operations; clicking on any of these items embeds the corresponding identifier in the text editor. This is intended to help the user recall elements in the current context, since they are otherwise hidden by the window.

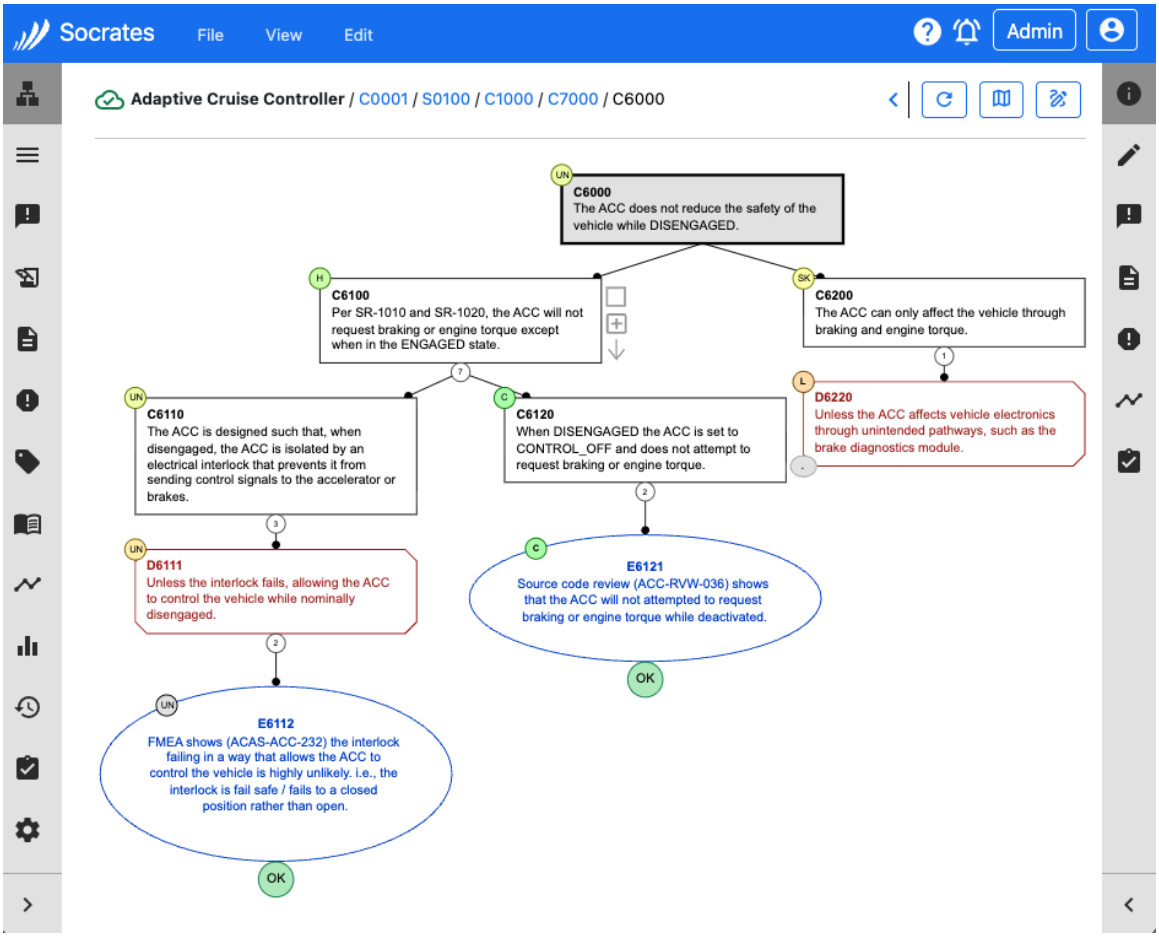


Figure 5.20: Visualizing belief assignments in *Socrates*.

Macros for Belief Assignments

Macros are an important feature of *Certus* because they encode commonly used belief assignment logic while also reducing the brittleness of the belief assessment under change. In *Socrates*, if the user opens the belief assessment window for a node with one or more children, they are provided an option to select one of the macros provided in the `certus-ts` implementation of the language. Macros appear in a dropdown list, as shown in Figure 5.23. The user may also opt to input a custom belief assignment, which reveals the text editor as shown in Figure 5.22.

Default Belief Assignments

The *Certus* language and its implementation in `certus-ts`, requires that every leaf node have a belief valuation provided in the `CertusCtx` and that every parent node

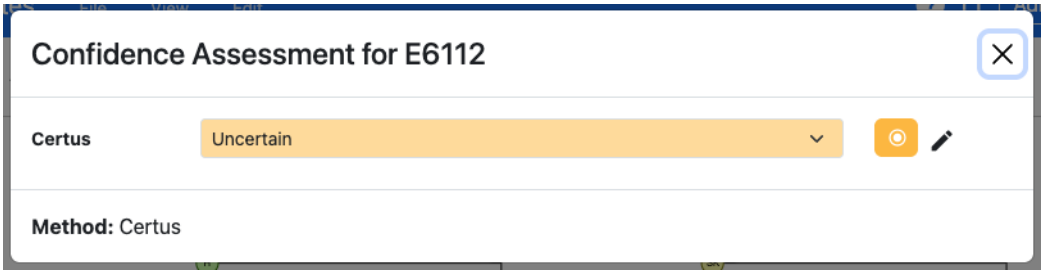


Figure 5.21: Assigning belief using canonical fuzzy sets in *Socrates*.

have a belief assignment operation specified. Otherwise, inputs are missing to the computation. In practice, this would require that users provide a full belief specification for an argument before computing even one belief assessment. This would severely limit the usability of the tool. Industrial scale ACs have hundreds or thousands of nodes.

As an alternative, the implementation of *Certus* in *Socrates* uses default belief assignments if a value is not provided by a user. Conservative, but reasonable, defaults are selected so that while the tool will produce *a* result, the user must still review each value in their own time to produce an accurate assessment. The defaults are as follows:

- **Premise-Typed Leaf Nodes.** For premise-type nodes at the leaves of the argument (usually Evidence or Solution nodes), the default assignment is *uncertain* (`UN`). This default takes the position that, when evidence is presented, unless otherwise indicated by the user, there is no reason to believe if the evidence is "good" or "bad".

- **Defeater-Typed Leaf Nodes.** For defeater-type nodes at the leaves of the argument, the default assignment is *certain* (`C`). This default takes the position that if a defeater is expressed in the AC, it is likely credible to some extent. The most conservative position to take is that it is certainly credible.

- **Parent Nodes.** When a node is a parent, the deductive `FUSE` macro is selected as the default operation for combining belief in the children to produce the parent. Though `MIN` is likely the more conservative choice, experience suggests that belief in most argument steps is more accurately modelled by `FUSE`. Further, while the strict macro variants might be more conservative than the deductive ones, the deductive interpretation is felt to align more closely with the average

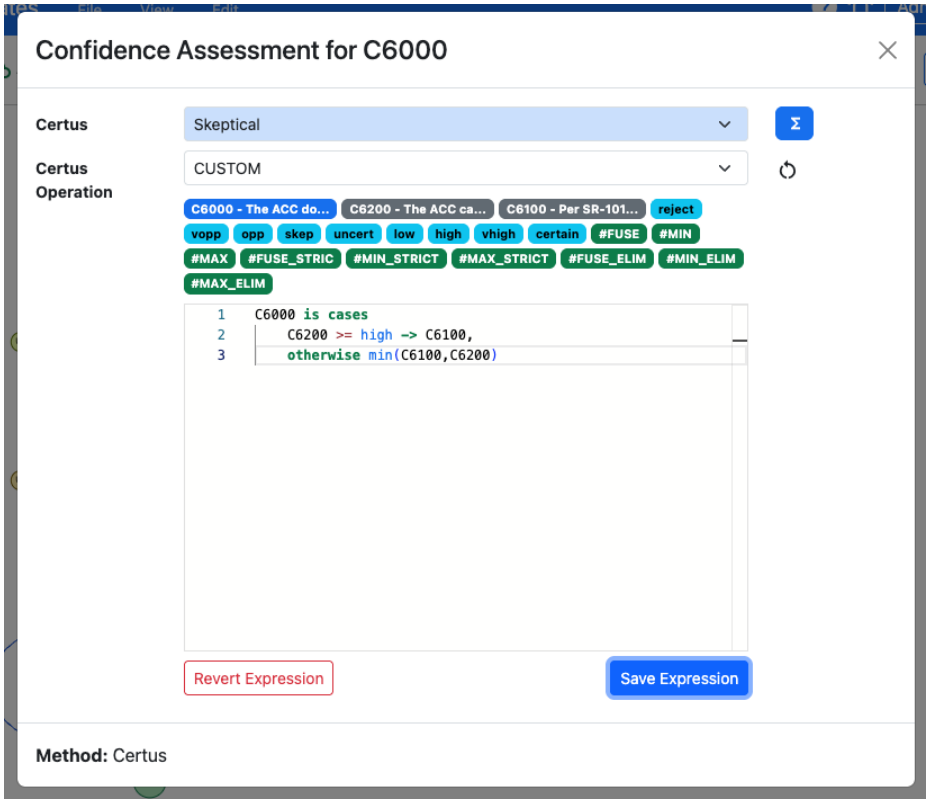


Figure 5.22: Authoring custom belief assignments in *Socrates*.

case while still offering a conservative perspective in terms of the impact of defeaters.

Leveraging Canonical Sets and Macros.

Belief assessment for an AC is a non-trivial undertaking, requiring a user to provide many inputs. This is true of all the quantitative CAMs, including, in the limit, *Certus*. However, the language and its integration in *Socrates* was specifically designed to reduce the number and complexity of inputs required by users for *most* cases. This property of the language will be analyzed more fully in Chapter 7.

For the time being, observe that the GUI in *Socrates* is designed to reduce the effort of a belief assessment if: 1) mainly the canonical fuzzy sets are used to input belief levels at the argument leaves, and 2) macros are used for belief assignments at parent nodes. More sophisticated belief assignment expressions remain possible, but should be reserved for exceptional cases.

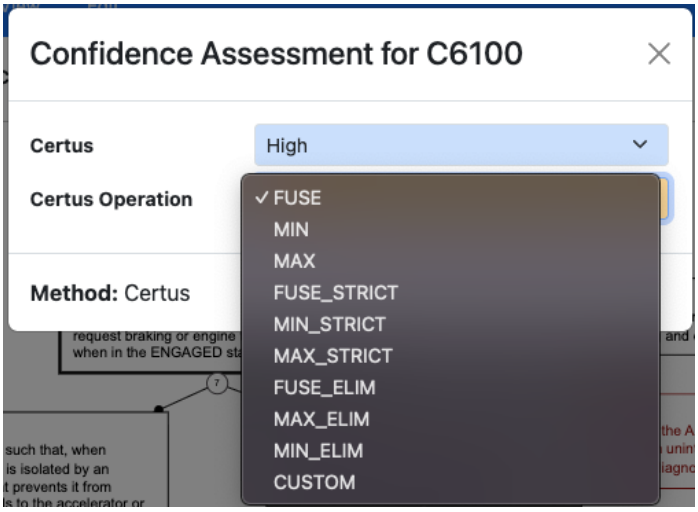


Figure 5.23: Selecting macros in *Socrates* for belief assignment.

5.9 Chapter Summary

This chapter has developed the *Certus* domain specific language, beginning from its theoretical foundations and ending with an implementation of the language that has been integrated into a commercial AC tool. *Certus* provides the benefits of a qualitative CAM, permitting users to reason linguistically about belief in an AC, and allows users to "look under the hood" and analyze belief quantitatively.

Theories for managing imperfect knowledge were surveyed. A combination of possibility theory and fuzzy set theory was selected for *Certus*. Their combined potential to mitigate subjectivity, enhance understandability, and handle dialectic reasoning made them a reasonable choice as a theoretical foundation for a CAM. However, none of the surveyed theories provided pre-defined operations that have the desired degree of expressivity for a CAM. This motivated *Certus* as a language for users to express more sophisticated belief propagation operations.

Using possibility and fuzzy set theory, belief in an AC claim was modelled using fuzzy sets over a joint possibility-necessity scale. The scale provides an interpretation of values (e.g., "X is possible, but not necessary") and the fuzzy sets allow for vague expressions of belief that cover multiple belief values. Canonical fuzzy sets were crafted to represent common belief levels that span the entire possibility-necessity scale and were given linguistically meaningful names (e.g., "certain" or "opposed").

The syntax and semantics of the *Certus* language were specified using the denotational approach, expressed using the Lean4 theorem prover's meta-programming

framework. The language's syntax is designed to be easy to understand, even for non-expert users. Language features such as user-defined operators and macros are intended to improve usability and reduce brittleness when the underlying AC is changed. Additional examples showing the application of the language appear in subsequent chapters and also in Appendix D and E.

The core specification for the language was extended to permit reasoning over indicators and artifacts associated with nodes in an AC. This enables belief assessment for dynamic assurance cases, that might change frequently as part of a continuous or perpetual assurance process.

Finally, the language specification was implemented in an open-source TypeScript library called `certus-ts`. As a proof of concept, this library was incorporated into a commercial AC tool, *Socrates*, including the development of graphical user interface elements aimed at improving "at-scale" efficiency. The implementation in *Socrates* will be used in subsequent chapters to demonstrate and evaluate *Certus*.

Chapter 6
The *Certus* Method

The previous chapter developed the *Certus* language as a means of describing how belief propagates through an AC's argument. The language also includes features for describing how belief is updated due to changes in a dynamic AC, such as when the availability of a supporting artifact changes or when the value of an indicator is updated. However, a language is merely a tool for expressing and organizing one's thinking. To be useful to practitioners, the language must be accompanied by a *method* that describes how to systematically use the language to assess and monitor belief in a dynamic assurance case.

This chapter defines the *Certus* method within the foundational Assurance Case Method described in Chapter 2. The method is applicable to both dynamic and static ACs. As defined, the method is intended to be applied in step with the Assurance Case Method, though it can also be adapted for other purposes. The adaptive cruise control (ACC) system argument fragment used in previous chapters is reused as a running example to illustrate the application of the method.

6.1 Overview of the *Certus* Method

The steps in the *Certus* method progressively develop, and then apply, a belief model for the AC. The method has ten steps. An important principle is that each argument step (including those involving evidence or defeaters) should be considered in isolation. This is intended to increase the objectivity of the assessment; for example, by reducing the temptation to "look ahead" and bias the assessment of the current argument step by considering descendant nodes in the argument[1].

[1]Admittedly, this is artificial, since users can look ahead if they want. Nonetheless, the recommendation is to consider each argument step in isolation.

1. **Model Belief Propagation** - For each argument step that is not supported by evidence, use *Certus* to describe how belief propagates from children to the parent. Include the impact of indicators as required.

2. **Model Support** - For each argument step that is supported by evidence, use *Certus* to describe how belief in a claim is impacted by the evidence. Include the impact of indicators as required.

3. **Establish Deployment Criteria** - Establish criteria that must be satisfied to deploy the system, including thresholds for belief in the top-level claim.

4. **Model Challenges** - For each defeater added to the argument that is rebutted by additional argumentation or evidence, use *Certus* to describe the impact of the defeater on the parent(s) and the impact of rebutting arguments and evidence on the defeater.

5. **Assess Challenges** - For each residual defeater, assign a belief valuation describing the level of credibility of the defeater. If indicators are used to monitor the defeater, then compose a *Certus* expression describing how they impact the defeater's credibility.

6. **Assess Evidence** - For each evidence node, assign a belief valuation for the evidence. If dynamic artifacts are used, then compose a *Certus* expression describing how the absence of the artifact impacts belief in the evidence.

7. **Decide on Deployment** - Apply the deployment criteria from Step 3 and decide whether (or not) to deploy the system. Return to system development if the AC cannot be deployed.

8. **Establish Operational Criteria** - Establish operational criteria that must be satisfied during deployment.

9. **Monitor Belief** - Monitor the belief levels in the argument by re-evaluating the *Certus* expressions in the argument when there is a change to an indicator's value to confirm that the operational criteria remain satisfied.

10. **Decide on Response** - If the operational criteria are not satisfied, then decide on the response, potentially revising the AC in response to change.

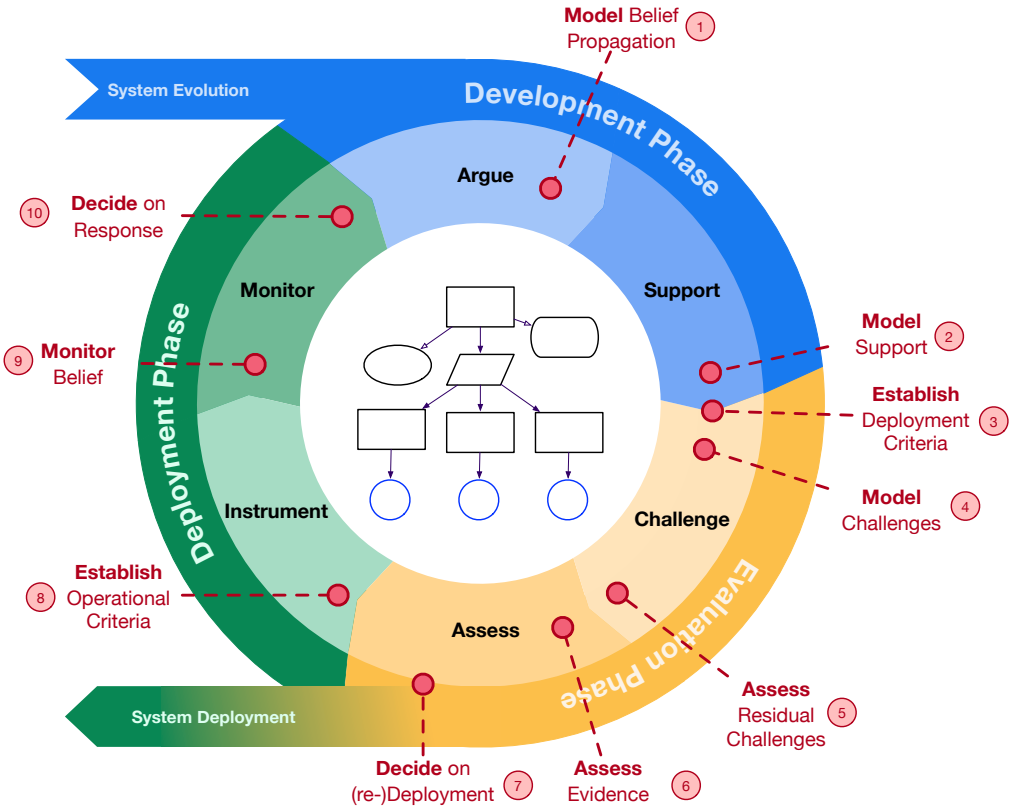


Figure 6.1: Overlaying the *Certus* method on the AC method from Chapter 2.

The *Certus* method is intended to be applied in step with the foundational AC method described in Chapter 2. Its steps are overlaid onto the phases of the AC method in Figure 6.1. The purpose of overlaying the methods, as opposed to developing a stand-alone method for *Certus*, is to aid adoption by practitioners. Indeed, one of the strategies for improving CAM adoption by practitioners is to *connect* the CAM to existing methods and processes that practitioners use [52]. Additionally, an overlaid method has the benefit of allowing users to concurrently develop the AC's argument and describe belief propagation at the same time so the argument and its belief model emerge together.

6.1.1 Ordering of the Steps

The ordering of the steps in the *Certus* method is a result of overlaying *Certus* on top of the AC method, which imposes a coarse ordering on the steps (e.g., Development

before Evaluation). Additionally, as noted above, an important principle when modelling and assessing belief is to consider each argument step in isolation. The ordering of the method's steps reflects a progression through the AC's argument structure: beginning with reasoning and concluding with an assessment of residual challenges and the credibility of the evidence.

Of note is the position of the "establish criteria" steps (Steps 3 and 8) in the method. These steps are intentionally placed at the beginning of the evaluation phase and deployment phase to avoid biasing assessments from subsequent activities impacting the selection of deployment and operational thresholds. For instance, if the deployment criteria were established after assessment of the evidence and residual challenges, then there might be pressure for analysts to select a threshold that is just below the computed belief valuation, undermining the purpose of having a threshold in the first place. However, the deployment threshold cannot be placed too early in development either, since there must be some argument in place over which to describe the threshold, and there should be some preliminary understanding of what satisfying, or failing to satisfy, the threshold entails.

Though the *Certus* method is described linearly, it is acceptable to revisit previous steps to account for changes to the argument or revise belief propagation operators. For instance, if Step 6 reveals that the available evidence is not satisfactory and new evidence is added to compensate, then it is necessary to return to Step 2 to model how the new evidence impacts its parent node(s).

6.1.2 Dynamic Assurance with the *Certus* Method

As with the foundational AC method, the *Certus* method is applicable in dynamic assurance processes. The role of *Certus* in each of the method's phases is as follows:

- **Re-Development** - When a DAC's argument is modified as part of system evolution or adaptation activities, changes to the argument mean that Steps 1 and 2 should be repeated. For instance, if a new type of evidence is provided, then *Certus* expressions for claims that depend on the new evidence might need to be updated. *Certus*'s macros are useful in this regard, as they are less brittle under argument change.

- **Re-Evaluation** - When a DAC changes, belief should be re-evaluated. This might involve reassessing evidence or raising new challenges against the ar-

gument. Steps 3 through 6 of the *Certus* method should be repeated. Re-evaluation should culminate in a (re-)deployment decision.

- **Re-Deployment** - During the Deployment phase, the system's operations are monitored and *Certus* is used to re-compute belief in the DAC for updated indicator values. If operational criteria are violated, then a decision must be made about how to respond (e.g., adapt or re-initiate development).

6.1.3 Tailoring the *Certus* Method

Though the *Certus* method is intended to be carried out in parallel with the AC method, situations might arise where the idealized description of the method does not align with the needs of a specific project or organization. In such cases, the *Certus* method can be tailored, as described by the following situations.

Applying *Certus* Retroactively. The need might arise to apply *Certus* retroactively, after an AC has been fully developed and connected to supporting evidence. In these scenarios, the steps of the *Certus* method should still be applied in the same order, but will be "squeezed" into a single step in the broader AC method. For example, Steps 1 through 6 should be applied as part of the AC method's *Assess* step, before a deployment decision is made.

Applying *Certus* Without Challenge. Some organizations might not employ dialectic reasoning as part of their AC development process. In this case, Steps 4 and 5 should be omitted.

Applying *Certus* to Static ACs. In practice, many ACs remain static and are not monitored at run-time as part of continuous or perpetual assurance processes. The *Certus* method may be tailored to remove the run-time aspects that are part of the Deployment phase, i.e., Steps 8 through 10.

6.2 *Certus* in the Development Phase

The *Certus* Method begins in the Development Phase with a focus on modelling how belief propagates through the premises in the argument. While other details of the argument might be known in advance, such as details of the supporting artifacts or prospective defeaters, they are set aside during the Development Phase. There are

two steps: 1) Modelling Belief Propagation, and 2) Modelling Support, which are discussed in turn below.

6.2.1 Step 1: Modelling Belief Propagation

For each step in the argument whose children are not evidence nodes, the *Certus* language is used to describe how belief propagates from children to a parent and to account for the impact of indicators on belief, if they are used. Belief propagation may be described by either a macro (e.g., `#FUSE`) or by composing a custom expression (e.g., using `cases`). In many cases, it might be acceptable to use the default propagation operation defined by the language's implementation. Belief modelling may be performed in a top-down or bottom-up manner, though each argument step should be considered in isolation.

For the macros available in the `certus-ts` implementation, the following applicability criteria may be used. The `#MIN` macro should be used when all children in an argument step must be true for the parent to be true. Conversely, the `#MAX` macro should be used when only one of the children must be true. Finally, the `#FUSE` macro should be used when belief in the parent should be computed as a balanced merge ("average") among the child beliefs. Since defeaters are not considered in this step, there is no need to distinguish between the deductive, eliminative, and strict dialectic interpretations. The deductive interpretation is recommended as a default.

Pre-Conditions for Step 1. The following conditions should be satisfied prior to starting Step 1: 1) the argument has at least one step that is not a leaf and not directly supported by evidence; and 2) if indicators are to be used, they are already defined and their relationship to claims in the argument is known.

Post-Conditions for Step 1. After completing Step 1, all reasoning steps in the argument that are not directly supported by evidence have a *Certus* expression describing how belief propagates from children to parents.

Worked Example for Step 1. In the running ACC example shown in Figure 6.2, two *Certus* propagation operations are defined in this step. For `C6000`, a `cases` expression is used to condition on the belief in `C6200` achieving a minimum threshold. For `C6100`, the `#FUSE` macro is chosen to merge child beliefs. No other reasoning steps in the argument fragment are applicable in Step 1. No indicators are applicable in the example for this step.

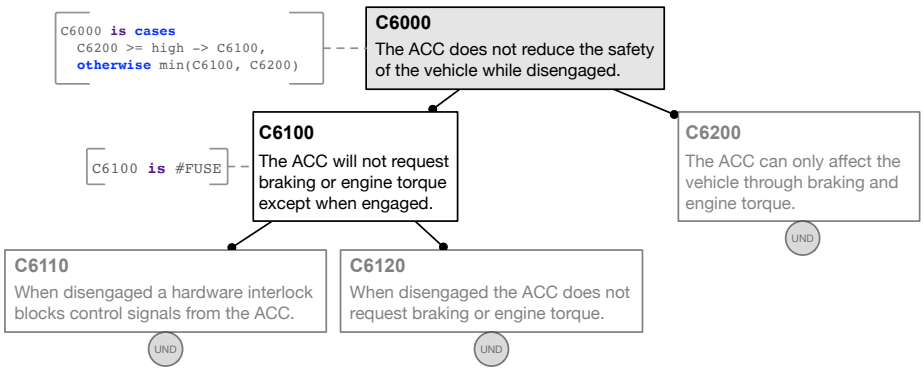


Figure 6.2: Example of Step 1 in the *Certus* Method.

6.2.2 Step 2: Modelling Support

In Step 2, *Certus* is used to describe how belief in evidence changes belief in the lowest level claims in the argument. If indicators are to be used, then they should also be incorporated in the propagation expressions. In this step, no assumptions should be made about the underlying artifacts. The propagation operators should account for all possible evidence valuations, ranging from certainty to rejection. As in Step 1, either macros or custom *Certus* expressions can be used to describe belief propagation.

Pre-Conditions for Step 2. The following conditions should be satisfied prior to starting Step 2: 1) the argument has at least one step that is supported by an evidence node; and 2) if indicators are to be used, they are defined and their relationship to the lowest level claims in the argument is known.

Post-Conditions for Step 2. After completing Step 2, all reasoning steps in the argument that are supported by an evidence node have a *Certus* expression describing how belief propagates from the evidence to the lowest level of claims.

Worked Example for Step 2. In Figure 6.3, two new *Certus* expressions are added to describe how evidence `E6112` and `E6121` impact belief in their respective parents. In both cases, the expressions bound the amount of belief that can propagate from child to parent, which is intended to account for inadequate evidentiary support. No indicators are applicable to the claims in the example for Step 2.

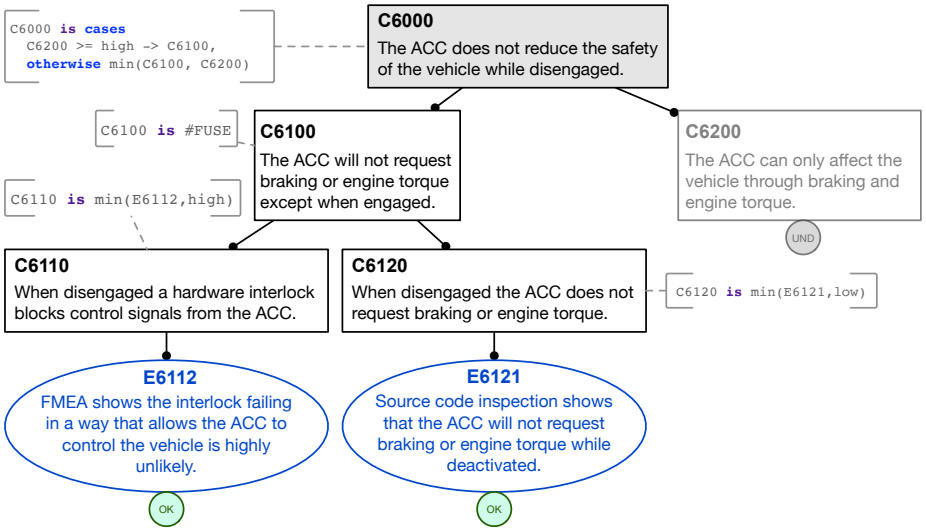


Figure 6.3: Example of Step 2 in the *Certus* Method.

6.3 *Certus* in the Evaluation Phase

In the Evaluation Phase, the *Certus* method is focused on assessing belief in the argument with the goal of making a decision to deploy the system. There are five steps in this phase. Though there is some fluidity in their order, it is important that Step 3 be performed early during evaluation to establish firm criteria for deployment.

6.3.1 Step 3: Establish Deployment Criteria

Before evaluating the AC, criteria should be established that must be satisfied prior to deploying the system. While numerous deployment criteria might exist for a system and its AC, from the perspective of the *Certus* method, the criteria should be formulated based on the level of belief for one or more nodes in the argument; for example: "belief in the top-level claim is at least *very high*." At this time, *Certus* does not contain a mechanism for expressing and evaluating deployment decision criteria. For the purpose of defining the *Certus* method, it is assumed that these criteria can be evaluated by a human supervisor. Since the criteria amount to a series of Boolean expressions, it is expected that their evaluation can be automated with modest effort.

Pre-Conditions for Step 3. Prior to starting Step 3, the argument should be developed to the point where the top-level claim has been supported by a combination of sub-claims and evidence.

Post-Conditions for Step 3. After completing Step 3 a list containing one or more deployment criteria should be available. Each criterion should be expressed as a Boolean condition about the belief in claim(s) in the argument.

Worked Example for Step 3. For the ACC example, since only a small argument fragment is considered, one deployment criterion is identified: "Belief in `C6000` must be at least *very high*."

Remarks on Choosing Belief Thresholds. An important question arises when selecting deployment criteria: *what is the minimum level of belief required for deployment?* There is no definitive and universal answer to this question, as it depends on a range of factors, including the risk associated with the system and the risk tolerance of the responsible organization(s). It is tempting to say that certainty is required for deployment. However, this ignores the realities of real-world engineering practice, where true certainty is rare. Moreover, if certainty becomes a target, then there is little room for experts to express doubt which might contribute to confirmation bias. Belief levels *very high* or *high* might be acceptable alternatives.

6.3.2 Step 4: Model Challenges

In the Challenge step of the foundational AC Method, defeaters are raised against claims or evidence in the argument. Step 4 parallels this activity. For each defeater added to the AC, the belief propagation expression for its parent should be reviewed, and might need to be updated. For custom expressions, this might mean changing the logic encoded in the expression. For reasoning steps that used *Certus*'s macros, the macro selection should be reviewed to confirm it remains valid. In particular, a choice must be made about whether the deductive, eliminative, or strict dialectic interpretation will be applied to the argument step.

Additional argumentation or evidence is often used to rebut defeaters during the Challenge step. When this occurs, *Certus* should be applied to these sub-arguments as well, following Steps 1 and 2 described above.

If the belief level for a non-leaf (i.e., non-residual) defeater depends on indicators, then they should be incorporated into the belief propagation expression during this step.

Pre-Conditions for Step 4. Prior to starting Step 4, the argument should contain at least one defeater-typed node. If no defeaters exist in the argument, then this

step should be skipped. If indicators are to be used, then they should be defined and associated to the relevant defeaters.

Post-Conditions for Step 4. After completing Step 4 all defeaters should be accounted for in *Certus* propagation expressions. If a defeater is itself the parent of a argument step, then it should have a belief propagation expression over its children. All defeaters should be either supported by sub-defeaters, rebutted by argument or evidence, or marked as residual.

Worked Example for Step 4. In Figure 6.4 the results of completing Step 4 for the ACC example are shown. A new defeater is added, `D6220`; since no rebuttal is provided, it is marked as residual. An expression to propagate belief from `D6220` to `C6200` inverts the defeater's belief and then bounds the belief in the parent for cases where the defeater has lower than `low` belief.

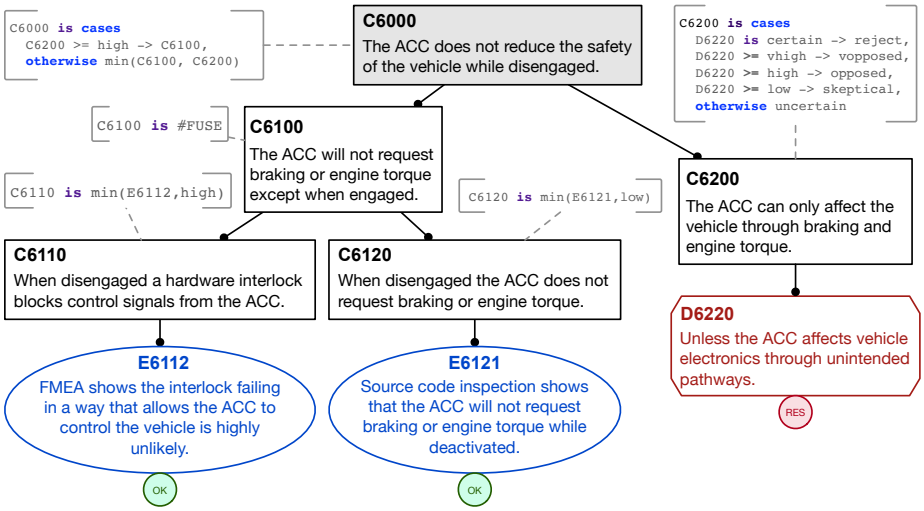


Figure 6.4: Example of Step 4 in the *Certus* method.

6.3.3 Step 5: Assess Challenges

Once defeaters have been incorporated into the belief propagation model, Step 5 provides a belief valuation for any residual defeaters that exist in the argument. A *Certus* expression should be provided that either: provides a direct belief assignment (e.g., `D1234 is low`) or uses `cases` to incorporate indicators. When valuing the belief in defeaters, the scale from Section 5.3 should be used to aid in interpretation. For instance, per Table 5.3, *Certus*'s canonical set *skep* (`SK`) corresponds to a situation where the defeater is more likely to be true than false.

Pre-Conditions for Step 5. Before beginning Step 5 the argument should have at least one residual defeater. If no residual defeaters exist, this step can be skipped. If indicators will be used, then they should be defined and associated to the corresponding residual defeaters.

Post-Conditions for Step 5. After completing Step 5 all residual defeaters should have a belief valuation, or their valuation should be computable using one or more indicators.

Worked Example for Step 5. The results of Step 5 for the ACC example are shown in Figure 6.5 below, along with the results from Steps 6 and 7. For the current step, observe that `D6220` has a *Certus* expression that incorporates the indicator `$events`, which corresponds to the number of field reports in the last month with suspected interactions between the AC and other vehicle systems. The indicator has a category called `warning`. When the corresponding threshold is exceeded the defeater is thought to be credible.

6.3.4 Step 6: Assess Evidence

In Step 6, the evidence-typed leaf nodes are assigned belief valuations corresponding to the belief in the trustworthiness of the underlying evidence. Assignments are made using either a direct belief assignment (e.g., `E1234 is certain`) or using custom *Certus* expressions that condition on the presence of artifacts or indicator values. In a continuous or perpetual assurance process, where artifacts are automatically generated as outputs from other activities, many evidence nodes might use custom expressions conditioned on the presence of an artifact.

Recall that in Step 2, the relationship between the evidence and its parent(s) was already modelled, so this Step 6 is focused strictly on the evidence itself. If evidence is described in a factual style, then it might be the case that most evidence nodes are assigned relatively high levels of belief. For instance, if the evidence node states "All Tests Passed", then it should be straightforward to assess the underlying evidence and confirm this fact is certainly true. Of course, this says nothing of whether the evidence is sufficient to support its parent, hence the need for modelling support in Step 2. If the trustworthiness of the evidence is in question, then undermining defeaters can also be raised to challenge the evidence. Continuing the testing example, perhaps there is reason to believe the test results were not tabulated correctly, which would undermine the credibility of the test results to support claims in the AC.

Pre-Conditions for Step 6. Prior to beginning Step 6, the argument should have at least one evidence-typed leaf node. If artifacts and indicators are used in the assessment, then they should also be defined and associated to their corresponding evidence node(s). If no evidence-typed leaf nodes are present in the argument, then this step should be skipped.

Post-Conditions for Step 6. After completing Step 6, all evidence nodes in the argument should either have a belief valuation, or an expression from which a valuation can be computed based on artifacts and indicators.

Worked Example for Step 6. In Figure 6.5 below, the two evidence nodes are assessed. Node `E6112` had its belief directly assigned. The belief in `E6121` is conditioned on whether the artifact `@inspectionReport` is available, indicated by the  icon.

6.3.5 Step 7: Decide on Deployment

In the final step of the Evaluation Phase, *Certus*'s evaluation procedure is used to compute the belief for each node in the argument. The results are then checked against the deployment criteria from Step 3 to determine if the system can be deployed.

Pre-Conditions for Step 7. Before starting Step 7, the following should be true: 1) there is at least one deployment criterion conditioned on the belief in a node in the argument; 2) all indicators used by *Certus* have an assigned value; 3) all reasoning steps have a *Certus* expression to propagate belief from children to parents; and 4) there are no undeveloped lines of reasoning in the argument.

Post-Conditions for Step 7. After completing Step 7, all nodes in the argument will have a belief valuation and, from the perspective of the AC, a decision will have been made to deploy (or not) the system. If the system cannot be deployed, then return to system development activities to address weaknesses in the AC before resuming the *Certus* method.

Worked Example for Step 7. The results of applying *Certus*'s evaluation procedure are shown in Figure 6.5. Each node is annotated with its computed belief level. It is assumed that the `events` indicator had a value of 9, which is less than its `warning` threshold, and that the `inspectionReport` artifact was available at the time of evaluation. The result is an *uncertain* (`UN`) level of belief in the top-level claim, which does not satisfy the criterion from Step 3: the system cannot be deployed.

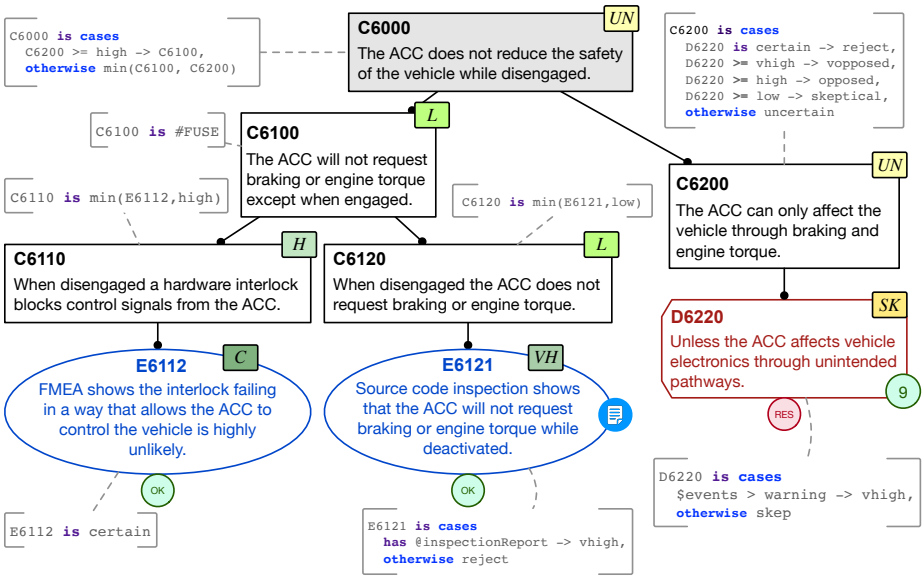


Figure 6.5: Example of Steps 5 to 7 in the *Certus* method.

Remark on Deployment Decisions. In practice, the approval of the AC is part of a larger deployment decision making process. Organizations deploying critical systems usually have additional criteria for deployment that are unrelated to the AC. For instance, the organization's legal team might be in the process of purchasing an insurance policy[2].

6.4 *Certus* in the Deployment Phase

In the Deployment Phase, the AC's argument and supporting evidence do not change, but changes to indicator values (measured from the deployed system) trigger re-evaluation of the AC's belief, and the results are used to decide whether the system can continue to operate. The *Certus* method has three steps in this phase.

6.4.1 Step 8: Establish Operational Criteria

Immediately after a decision to deploy the system is made, criteria should be defined for the continued operation of the system. This step mirrors Step 3, where deployment criteria were defined. Indeed, the deployment and operational criteria might be the same.

[2] "Just in case."

Pre-Conditions for Step 8. Prior to beginning Step 8, the decision should have been made to deploy the system. If the decision is not to deploy, this step is not applicable.

Post-Conditions for Step 8. When this step is complete, the operational criteria will be specified.

Worked Example for Step 8. The ACC example used above resulted in the decision not to deploy the system because the belief in the top-level claim was insufficient. However, suppose the belief had been *very high* and the decision was made to deploy the system. The operational criteria might be: "The belief in the top-level claim remains at least *very high*."

6.4.2 Step 9: Monitor Belief

During system operation, the system is monitored and the values of indicators are updated in the AC. The belief levels are re-evaluated for each change to the indicator(s) and the operational criteria re-checked. The *Certus* method "remains" in this step until the operational criteria are no longer satisfied or if the system is taken out of service.

Pre-Conditions for Step 9. Prior to beginning Step 9, the indicators that are instrumenting the AC should be connected to data sources that are updated regularly as part of system operations. If there are no indicators, then this step is not applicable.

Post-Conditions for Step 9. When this step is completed, the operational criteria are no longer satisfied or system operations have ceased.

Worked Example for Step 9. Suppose that the value of the `events` indicator increases from 9 events per month to $>$ 10 events per month, which exceeds the warning threshold given in Figure 6.5. This would trigger `D6220` to have *very high* belief resulting in the top-level claim being *very opposed* and violating the operational criteria.

6.4.3 Step 10: Decide on Response

In the final step of the method, a decision must be made about how to respond to unsatisfied operational criteria because there is insufficient belief in the AC's claim(s). One option is to take the system out of service. However, as with the decision to

deploy, the decision to continue to operate a system might involve factors beyond the scope of the AC. It might be necessary to continue to operate a potentially unsafe system if the overall consequences of ceasing operations are worse than the consequences of loss events contemplated by the AC[3]. Notably, depending on the type of system, the decision might be made by a human supervisor or as part of an automated process. For instance, in a self-adaptive system, the decision, and subsequent response, might be made by the MAPE-K's Analyzer component and then carried out by the Planner and Executor components.

Pre-Conditions for Step 10. Prior to starting this step, the system must be operating and operational criteria for the AC must not be satisfied.

Post-Conditions for Step 10. After this step is completed, a decision will have been made about how to respond to the insufficient belief in the AC.

Worked Example for Step 10. For the ACC example, a reasonable response for the vehicle manufacturer might be as follows. First, investigate the change in the `events` indicator. Next, develop a software patch that addresses the underlying issue and repeat the AC and *Certus* methods to produce an updated AC for the new version of the system. Finally, release the change as part of their vehicle recall program.

6.5 Chapter Summary

This chapter has defined the *Certus* method, which applies the *Certus* language to evaluate belief in a dynamic assurance case. The method has ten steps that are applied in the respective phases of the Assurance Case Method described in Chapter 2. Each step of the *Certus* method was defined with pre- and post-conditions and supported by a small worked example. In the Development Phase, *Certus* is used to model belief propagation and evidentiary support. In the Evaluation Phase, evidence and defeaters are assessed, and a deployment decision is made based on pre-defined deployment criteria. Indicators and artifacts may be incorporated into *Certus*'s expressions to make the overall belief in the AC sensitive to contextual and operational factors. Finally, in the Deployment Phase, changes to indicators trigger re-evaluation of the AC's belief, and if operational criteria are not satisfied, then a decision must be made about how to proceed.

[3]For example, the consequence of a hospital closing its doors might be worse than continuing to operate, even if the AC for the hospital's health information system has been invalidated.

Chapter 7
Evaluating *Certus*

The previous chapters proposed the *Certus* language and method as a means of assessing belief in a static or dynamic AC. The definition of the language and method was accompanied by worked examples to illustrate key concepts. While examples are helpful, they are not a systematic evaluation of *Certus*' properties and capabilities. This chapter presents the results of a series of evaluation activities that investigate properties of *Certus* and demonstrate its capability with a case study.

Four analyses will investigate key properties of the language: 1) belief *propagation* and susceptibility to attenuation or amplification effects; 2) its *scalability* in terms of end-user effort as the argument grows in size; 3) the *sensitivity* of *Certus*' macros to changes in belief; and 4) the ability of the language to *express* belief propagation operations. Each of these analyses considers a property in isolation. To evaluate the end-to-end capability of *Certus*, including the applicability to dynamic ACs, this chapter also contains a case study that applies *Certus* to an AC developed for a real-world (prototype) medical device. The chapter closes with a review of the CAM requirements defined in Chapter 4 that examines whether each requirement is satisfied by the combined theoretical and empirical evidence presented thus far.

7.1 Propagation Analysis

A known problem with existing quantitative CAMs is that, as the argument's structure grows in either depth or breadth, the belief in parent claims can be impacted in unintuitive ways. For example, a deep argument composed of conjunction operators that has strong, but imperfect, support at its leaves might have a significantly *attenuated* belief in the argument's top-level claim. This occurs due to successive

multiplications of fractional numbers as belief propagates upward through the argument. The opposite phenomenon can occur when disjunctions are the dominant propagation operator, which can result in belief *amplification*. In either case, the outcome is undesirable because it results in an inaccurate and potentially unintuitive belief assessment.

Problems of this nature have been observed by various authors. Idmessaoud et al. investigated the belief amplification and attenuation properties of their DST method [50]. Graydon and Holloway also identified this problem, framing it as "sensitivity to the arbitrary scope of hazards" [80]. They performed a test where they replaced a singular claim (about a hazard being mitigated) with many claims (sub-hazards being mitigated), each with the same belief, and found unexpected changes in belief levels that did not align with their intuition [80].

This section evaluates the susceptibility of *Certus* to attenuation and amplification effects encountered during belief propagation by other quantitative CAMs. Since users can write case-wise belief propagation expressions in *Certus*, which are immune to these effects, the focus is placed on *Certus*' macros which are closer in nature to calculations of other quantitative CAMs. A series of tests are used to probe *Certus*' macros for belief attenuation and amplification. The tests are repeated using Hobbs and Lloyd's BBN method and Idmessaoud et al.'s DST method to provide a point of comparison.

7.1.1 Method

To assess the effect of belief attenuation and amplification, two sets of test cases were designed, one for argument depth and another for argument breadth.

Test Cases for Argument Depth

For argument depth, a series of binary tree argument structures were used where the trees had a line of reasoning that increased in depth through the series, as shown in Figure 7.1. The cases are labelled *Depth 1* through *Depth 5*, where the label number corresponds to the depth of the longest chain of claims (black shaded circles) in the argument. Each line of reasoning terminates with a single evidence node (blue unshaded circles). Belief valuations will be provided for each evidence node.

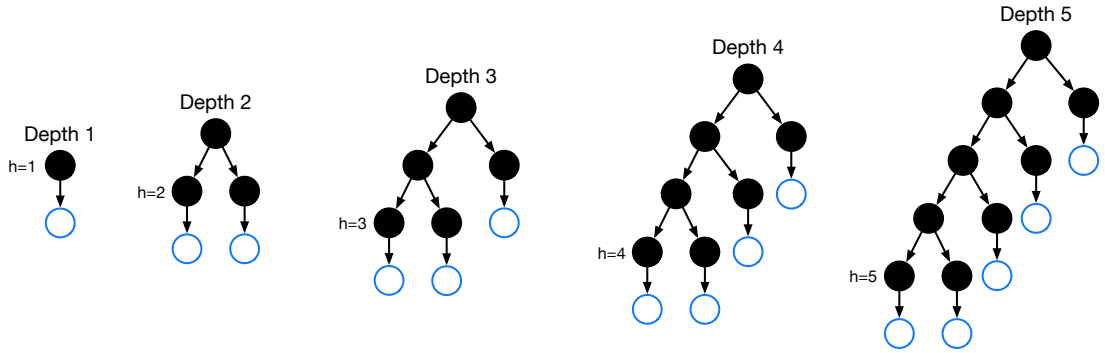


Figure 7.1: Depth test cases for propagation analysis.

Test Cases for Argument Breadth

For argument breadth, a series of shallow argument structures containing a single argument step were created. Each argument adds another child to the argument step, as depicted in Figure 7.2. The cases are labelled *Breadth 1* through *Breadth 5*. The notation is the same as for the argument depth test cases above.

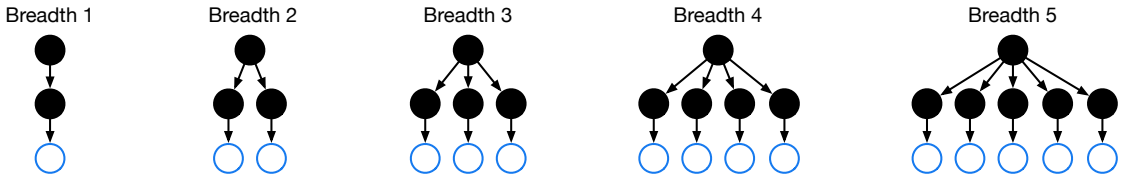


Figure 7.2: Breadth test cases for propagation analysis.

The breadth cases are similar to the tests proposed by Graydon and Holloway to evaluate changes to the ("arbitrary") scope of hazards [80]. To see this, let the single claim in *Breadth 1* correspond to the root of a sub-argument showing that one broadly defined hazard is mitigated. Then subsequent cases correspond to situations where the single hazard is split into separate hazards each with smaller scope. Graydon and Holloway argue that, if the belief in the evidence supporting the mitigation of a hazard is the same as the belief once the hazard is split, then the belief in the top-level claim should not change. For example, consider a scenario where the belief supplied for the evidence node in *Breadth 1* is *very high* (respectively, 0.9). Then *Breadth 4* is the situation where the hazard is split into four hazards of smaller scope. If the belief supplied for each evidence node in *Breadth 4* is also *very high*, then the belief in the top-level claim should remain the same, since nothing has fundamentally changed about the belief in the effectiveness of the mitigations.

Analysis Procedure

Belief computations were performed with the implementations of each CAM provided by the *Socrates* tool. The depth and breadth test sets defined above were repeated for combinations of leaf belief valuations and CAM belief propagation configurations as shown in Table 7.1. Three leaf belief valuations were used: *very opposed*, *uncertain*, and *very high*. Comparable valuations were supplied for the BBN and DST methods: 0.1, 0.5, and 0.9 for the BBN method and the DST's "decision" scale, with DST's "confidence" 1.0.

Table 7.1: Configurations tested in the propagation analysis.

#	**Method**	**Operator**	**Parameters**
1	*Certus*	#FUSE	–
2	*Certus*	#MIN	–
3	*Certus*	#MAX	–
4	BBN	NoisyAND	$w_i = 1.0,\ k = 0$
5	BBN	NoisyOR	$w_i = 1.0,\ k = 0$
6	BBN	NoisyAND	$w_i = \frac{1}{n},\ k = 0$
7	BBN	NoisyOR	$w_i = \frac{1}{n},\ k = 0$
8	DST	C-Arg	$w_f = w_r = 1$
9	DST	D-Arg	$w_f = w_r = 1$
10	DST	H-Arg	$w_f = w_r = 1$
11	DST	C-Arg	$w_f = w_r = \frac{1}{n}$
12	DST	D-Arg	$w_f = w_r = \frac{1}{n}$
13	DST	H-Arg	$w_f = w_r = \frac{1}{n}$

For *Certus*, the deductive macros are used without loss of generality. The macros behave identically among the three dialectic interpretations if there are no defeaters present, and this set of tests does not consider defeaters.

7.1.2 Results

The detailed results of the propagation analysis are shown in Tables 7.2 and 7.3 for the depth and breadth sets of test cases, respectively. Several observations about the results are provided below.

Table 7.2: Results of **depth** test cases for propagation analysis for operators in the BBN, DST, and *Certus* methods. For DST results, two values are provided, the "decision" score followed by the confidence score in parentheses. For *Certus*, three canonical sets are used: *very high* (VH), *uncertain* (UN), and *very opposed* (VO).

		BBN ($w_i = 1$)		**BBN** ($w_i = 0.5$)		**DST** ($w_f = w_r = 1$)			**DST** ($w_f = w_r = 0.5$)			***Certus***		
Belief	h	AND	OR	AND	OR	C-Arg	D-Arg	H-Arg	C-Arg	D-Arg	H-Arg	FUSE	MIN	MAX
VH (0.9)	**1**	0.9000	0.9000	0.9000	0.9000	0.9000 (1)	0.9000 (1)	0.9000 (1.00)	0.9000 (1.00)	0.9000 (1.00)	0.9000 (1.00)	VH	VH	VH
	2	0.8100	0.9900	0.9045	0.6975	0.8100 (1)	0.9999 (1)	0.9000 (0.82)	0.8325 (0.96)	0.8438 (0.71)	0.8775 (0.91)	VH	VH	VH
	3	0.7290	0.9990	0.9043	0.6418	0.7290 (1)	0.9990 (1)	0.9360 (0.91)	0.8051 (0.85)	0.8204 (0.64)	0.8888 (0.88)	VH	VH	VH
	4	0.6561	0.9990	0.9037	0.6265	0.6561 (1)	0.9999 (1)	0.9360 (0.89)	0.7748 (0.76)	0.8132 (0.63)	0.8944 (0.88)	VH	VH	VH
	5	0.5905	1.0000	0.9025	0.6223	0.5905 (1)	1.0000 (1)	0.9392 (0.90)	0.7450 (0.69)	0.8111 (0.62)	0.8972 (0.88)	VH	VH	VH
UN (0.5)	**1**	0.5000	0.5000	0.5000	0.5000	0.5000 (1)	0.5000 (1)	0.5000 (1.00)	0.5000 (1.00)	0.5000 (1.00)	0.5000 (1.00)	UN	UN	UN
	2	0.2500	0.7500	0.5625	0.4375	0.2500 (1)	0.7500 (1)	0.5000 (0.50)	0.3125 (0.88)	0.5938 (0.69)	0.4375 (0.75)	UN	UN	UN
	3	0.1250	0.8750	0.5859	0.4141	0.1250 (1)	0.8750 (1)	0.5000 (0.75)	0.3203 (0.61)	0.6445 (0.54)	0.4688 (0.69)	UN	UN	UN
	4	0.0625	0.9375	0.5947	0.4053	0.0625 (1)	0.9375 (1)	0.5000 (0.63)	0.3154 (0.49)	0.6714 (0.47)	0.4844 (0.67)	UN	UN	UN
	5	0.0313	0.9688	0.5980	0.4020	0.0313 (1)	0.9688 (1)	0.5000 (0.69)	0.3097 (0.44)	0.6854 (0.43)	0.4922 (0.67)	UN	UN	UN
VO (0.5)	**1**	0.1000	0.1000	0.1000	0.1000	0.1000 (1)	0.1000 (1)	0.1000 (1.00)	0.1000 (1.00)	0.1000 (1.00)	0.1000 (1.00)	VO	VO	VO
	2	0.0100	0.1900	0.3025	0.0975	0.0100 (1)	0.1900 (1)	0.1000 (0.82)	0.0325 (0.96)	0.1438 (0.91)	0.0775 (0.91)	VO	VO	VO
	3	0.0010	0.2710	0.3582	0.0963	0.0010 (1)	0.2710 (1)	0.0640 (0.91)	0.1456 (0.71)	0.1837 (0.18)	0.0888 (0.90)	VO	VO	VO
	4	0.0001	0.3439	0.3735	0.0957	0.0001 (1)	0.3439 (1)	0.0640 (0.89)	0.1774 (0.65)	0.2198 (0.75)	0.0944 (0.89)	VO	VO	VO
	5	0.0000	0.4095	0.3777	0.0955	0.0000 (1)	0.4095 (1)	0.0608 (0.90)	0.1863 (0.63)	0.2525 (0.69)	0.0972 (0.88)	VO	VO	VO

Table 7.3: Results of **breadth** test cases for propagation analysis for operators in the BBN, DST, and *Certus* methods. For DST results, two values are provided, the "decision" score followed by the confidence score in parentheses. For *Certus*, three canonical sets are used: *very high* (VH), *uncertain* (UN), and *very opposed* (VO).

		BBN ($w_i = 1$)		**BBN** ($w_i = 1/n$)		**DST** ($w_f = w_r = 1$)			**DST** ($w_f = w_r = 1/n$)			***Certus***		
Belief	w	AND	OR	AND	OR	C-Arg	D-Arg	H-Arg	C-Arg	D-Arg	H-Arg	FUSE	MIN	MAX
VH (0.9)	**1**	0.9000	0.9000	0.9000	0.9000	0.9000 (1)	0.9000 (1)	0.9000 (1.00)	0.9000 (1.00)	0.9000 (1.00)	0.9000 (1.00)	VH	VH	VH
	2	0.8100	0.9900	0.9025	0.6975	0.8100 (1)	0.9900 (1)	0.9000 (0.82)	0.8325 (0.96)	0.8438 (0.71)	0.9000 (0.91)	VH	VH	VH
	3	0.7290	0.9990	0.9042	0.6526	0.7290 (1)	0.9990 (1)	0.8640 (0.73)	0.8051 (0.85)	0.8258 (0.65)	0.8879 (0.87)	VH	VH	VH
	4	0.6561	0.9999	0.9037	0.6392	0.6561 (1)	0.9999 (1)	0.8280 (0.66)	0.7748 (0.76)	0.8196 (0.64)	0.8753 (0.84)	VH	VH	VH
	5	0.5905	1.0000	0.9039	0.6293	0.5905 (1)	1.0000 (1)	0.7952 (0.59)	0.7450 (0.69)	0.8146 (0.63)	0.8633 (0.81)	VH	VH	VH
UN (0.5)	**1**	0.5000	0.5000	0.5000	0.5000	0.5000 (1)	0.5000 (1)	0.5000 (1.00)	0.5000 (1.00)	0.5000 (1.00)	0.5000 (1.00)	UN	UN	UN
	2	0.2500	0.7500	0.5625	0.4375	0.2500 (1)	0.7500 (1)	0.5000 (0.50)	0.3125 (0.88)	0.5938 (0.69)	0.5000 (0.75)	UN	UN	UN
	3	0.1250	0.8750	0.5822	0.4178	0.1250 (1)	0.8750 (1)	0.5000 (0.25)	0.3203 (0.61)	0.6464 (0.54)	0.5000 (0.64)	UN	UN	UN
	4	0.0625	0.9375	0.5862	0.4138	0.0625 (1)	0.9375 (1)	0.5000 (0.13)	0.3154 (0.49)	0.6757 (0.48)	0.5000 (0.58)	UN	UN	UN
	5	0.0313	0.9688	0.5905	0.4095	0.0313 (1)	0.9688 (1)	0.5000 (0.06)	0.3097 (0.44)	0.6891 (0.44)	0.5000 (0.55)	UN	UN	UN
VO (0.1)	**1**	0.1000	0.1000	0.1000	0.1000	0.1000 (1)	0.1000 (1)	0.1000 (1.00)	0.1000 (1.00)	0.1000 (1.00)	0.1000 (1.00)	VO	VO	VO
	2	0.0100	0.1900	0.3025	0.0975	0.0100 (1)	0.1900 (1)	0.1000 (0.82)	0.0325 (0.96)	0.1438 (0.91)	0.1000 (0.91)	VO	VO	VO
	3	0.0010	0.2710	0.3474	0.0958	0.0010 (1)	0.2710 (1)	0.1360 (0.73)	0.1456 (0.71)	0.1834 (0.82)	0.1121 (0.87)	VO	VO	VO
	4	0.0001	0.3439	0.3608	0.0963	0.0001 (1)	0.3439 (1)	0.1720 (0.66)	0.1774 (0.65)	0.2201 (0.75)	0.1248 (0.84)	VO	VO	VO
	5	0.0000	0.4095	0.3707	0.0961	0.0000 (1)	0.4095 (1)	0.2048 (0.59)	0.1863 (0.63)	0.2528 (0.69)	0.1367 (0.81)	VO	VO	VO

Effect of Propagation on *Certus* Macros

In both the depth and breadth test cases, *Certus*' `#FUSE`, `#MIN`, and `#MAX` macros did not exhibit any belief attenuation or amplification effects. For instance, when the leaf valuations were set to `VH`, regardless of the height or width of the argument, the computed belief value remains `VH` for all macros. In the case of `#MIN`, since all belief levels are set to `VH`, then the minimum belief is still `VH`. Similar outcomes are observed for other combinations of belief levels and macros.

Effects of Propagation on the BBN Method

In the left-most column of results in Tables 7.2 and 7.3, the NoisyAND ("AND") operator, in combination with link probabilities $w_i = 1$, results in belief attenuation for the BBN method. This is the most pronounced in the test with $h = 5$ and 0.9 assigned at the leaves: the *very high* belief in the evidence is reduced to just under 0.6, which is approaching uncertainty. The same result exists for the breadth cases. Belief attenuation is also observed for the NoisyOR ("OR") operator when the link probabilities are taken as $w_i = 1/n$. From Section 2.3.2, for NoisyOR the link probability defines the belief in the parent in the case where only that child is true and all other children are false. Since both children have $w_i < 1$, only a fraction of the belief from each child contributes to the parent, resulting in lower parent belief.

In the second data column in Tables 7.2 and 7.3, the NoisyOR operator, in combination with all $w_i = 1$, results in belief amplification. For instance, for the cases where the leaf valuation was taken as 0.5, having *uncertain* belief in all supporting evidence is amplified to 0.96, which is near certainty in the top-level claim. Amplification is also observed in the results for the combination of NoisyAND and $w_i = 1/n$. Recall that, for the NoisyAND operator, the link probabilities represent the reduction in belief in the parent claim if only that child is false and all other children are true. Since $w_i < 1$, this results in the "false belief" from each child being discounted, resulting in the parent belief increasing.

Effects of Propagation on the DST Method

When the forward and reverse parameters are taken as $w_f = w_r = 1$, the DST method produces the same results for its "C-Arg" and "D-Arg" as the BBN method's NoisyAND and NoisyOR operators, respectively. When the forward and reverse parameters are changed, the two methods produce different results, though the belief

attenuation and amplification effects persist. For instance, attenuation is present for C-Arg with $w_f = w_r = 0.5$ in both Tables 7.2 and 7.3.

The DST method defines an additional "H-Arg" operator, which is seen to be somewhat resistant to attenuation and amplification for its "decision" measure. This effect is visible for test cases with leaf valuations of 0.5. However, preserving the decision during propagation comes at the cost of reductions in the "confidence" measure, which is shown in parentheses in the tables.

7.1.3 Discussion

From the results above, *Certus*' macros do not appear to be susceptible to the attenuation or amplification effects that occur during belief propagation in the BBN and DST methods. This is not surprising given the underlying intention and definition of *Certus*' macros. For instance, the `#MIN` and `#MAX` macros are intended to propagate the minimum and maximum belief levels directly. This property of *Certus*' macros is desirable. It means that the belief valuations provided at the leaves of the argument are faithfully propagated through the argument, yielding a belief in the top-level claim that reflects the strength of the supporting evidence. Of course, the belief propagation *can* be attenuated or amplified in *Certus* with customized belief assignment expressions. But, this must be done intentionally by the user, and does not occur as a by-product of a formula.

There were no non-trivial cases where the BBN method did not exhibit some manner of belief attenuation or amplification. This could be taken as a desirable behaviour of the method in the sense that it shows the consequences of less-than-certain leaf valuations. Moreover, each argument step may be seen as a reasonable interpretation of how belief propagates from children to parents. For instance, the NoisyAND operator propagates the shared belief level among the children, and NoisyOR propagates the combined belief. In isolation, these seem like reasonable operators. However, when scaled over multiple argument steps the results can become problematic for at least two reasons. First, the belief in the top-level claim can differ significantly from the belief valuations supplied for the evidence such that the results do not align with intuition about the support for the argument. This reduces the understandability of the analysis results (see requirement R5). Second, the belief in the top-level claim is a function of depth and breadth of the argument meaning that the impact of modifying the argument might have unwanted effects on the overall assessment result (e.g.,

adding another layer of reasoning, or re-scoping claims into sub-claims).

For the DST method, several test cases produced attenuation and amplification effects. The results obtained by this analysis are consistent with those of Idmessaoud et al. who observed belief attenuation and amplification for the C-Arg and D-Arg operators and improved stability of the H-Arg operator [50]. The results are also consistent with Graydon and Holloway's finding about susceptibility to the "arbitrary scope of hazards", as seen from the breadth test cases [80].

Limitations

This propagation analysis has several limitations that impact how the results should be interpreted.

Limitations Due to Choice of Leaf Valuations. To bound the scope of the analysis, it was necessary to pick belief levels to be supplied at the leaves of the argument. Also, leaf valuations were fixed for each test, which was done to avoid confounding factors that might make the results more difficult to interpret. This setup is somewhat artificial, since a real-world AC is unlikely to have the same belief valuations for all leaf nodes. The analysis is limited in that it did not cover all possible combinations of leaf assignments. However, the goal of the analysis was to determine the effects of the propagation operators on belief, not to measure sensitivity to leaf valuations. So, selecting three different belief levels and using them consistently as leaf valuations is acceptable. Additionally, inspection of the definitions for propagation operators confirms that similar attenuation and amplification effects would be observed if more variable leaf valuations had been used.

Limitations Due to Operator Configurations. For the BBN and DST methods, only two configurations for each operator were studied. It is possible that different parameter configurations would mitigate attenuation and amplification effects. However, the configurations selected are reasonable. The first, $w_i = w_f = w_r = 1$, corresponds to a situation where each child has a maximum impact on its parent and is a conservative default for an analysis. The second $w_i = w_f = w_r = 1/n$ represents the situation where the impact on the parent is evenly distributed among the children, a common scenario in AC arguments. Even if ideal configurations exist, they are not described in [48, 50]. The chosen configurations for this analysis are reasonable configurations that a practitioner might apply without further guidance.

Limitations Due to Numerical Precision in Results. Some results from this analysis are subject to limitations in numerical precision due to rounding. For instance, in the left-most column of Tables 7.2 and 7.3, the bottom result (VO, $h = w = 5$) is reported as 0.0000. In fact, this should be 10^{-5}, but was reported as 0 due to rounding. This limitation is mentioned for completeness and does not impact the overall findings of the analysis.

Limitations Related to Defeaters. This propagation analysis has only considered arguments composed of premise-typed nodes (claims, evidence, etc.), not defeaters. It is not clear how defeaters could be included in this analysis, since they represent points of inversion of belief in the argument, which could be viewed as maximal belief attenuations. Including defeaters in the test cases would complicate results without clear benefit. The impact of defeaters on belief propagation will be studied in detail as part of a sensitivity analysis in Section 7.3.

7.1.4 Conclusion

Based on the results presented above, and accounting for the analysis's limitations, when applied to premise-typed argument steps, *Certus*' macros are not subject to the same belief attenuation and amplification effects that impact the BBN and DST methods. This property of *Certus* ensures that belief valuations supplied by users at the leaves of an argument are faithfully propagated to the top-level claim; and, if they are modified, it is a result of an intentional choice made by the user.

The attenuation and amplification effects observed for the BBN and DST methods mean that the overall confidence assessment is a function of leaf valuations, and also the argument's depth and breadth. This might produce unintuitive or difficult to understand results for novice users of these methods.

7.2 Scalability Analysis

One of the barriers to adopting CAMs identified in the Practitioner Perspective in Chapter 3 was that they require additional effort to apply. For quantitative and mixed CAMs, this usually means configuring belief computations with several inputs. However, given the size of AC arguments and the number of inputs required per argument step, configuring the computations can be a significant undertaking. Moreover, providing inputs is not simply a matter of blindly typing numbers into a tool, they

must be thoughtfully selected so that the CAM's computations align with the reasoning in the argument. Requirement R9 addresses this concern by requiring that a CAMs limit the necessary effort to apply them in the average or most common cases.

In the analysis of quantitative CAMs, Graydon and Holloway observed an absence of scalability data or analyses for quantitative CAMs [80]. They remarked that "it is not clear that [the surveyed] techniques requiring a substantial effort for each [argument step] will be feasible in practice ... the selected papers present no empirical evidence of scalability" [80].

To address R9 and in response to Graydon and Holloway's observation, this section performs a scalability analysis. The objective is to determine how *Certus*, as compared to two other quantitative CAMs, scales in terms of the number of inputs that must be made in its application in both the worst and average cases. Hobbs and Lloyd's BBN method and Idmessaoud et al.'s DST method are used for comparison [48, 50]. Borrowing from theoretical computer science, the analysis method is similar to time or space complexity analysis used for algorithms, but is referred to as the *decision complexity* of the CAM, in reference to the number of decisions that must be made during its application.

7.2.1 Method

When using a quantitative CAM it is necessary for the user to provide two types of inputs: 1) *leaf valuations* that assign belief at the leaves of the argument, usually to evidence nodes or residual defeaters; and 2) *step configurations* that describe how belief propagates from children to their parent at each argument step. Different quantitative CAMs have different approaches to specifying both these inputs. For example, the BBN method requires three parameters to configure each argument step whereas *Certus* requires a single belief assignment expression.

For this analysis, an argument is modelled as an n-ary tree of claims with height $h > 1$, where each leaf claim has m child evidence nodes. While real ACs have much more diverse structures, this admittedly artificial model of argument simplifies the analysis while also permitting meaningful comparison among quantitative CAMs. This model is visualized in Figure 7.3 for a claim tree with $h = 3$.

The n-ary tree of claims has a total of $n^h - 1$ nodes, each being a parent whose belief is computed based on $n \in \mathbb{N}$ children. Let $p \in \mathbb{N}$ be the number of decisions that a user of a CAM must make for each parent node. The h^{th} layer of the claim

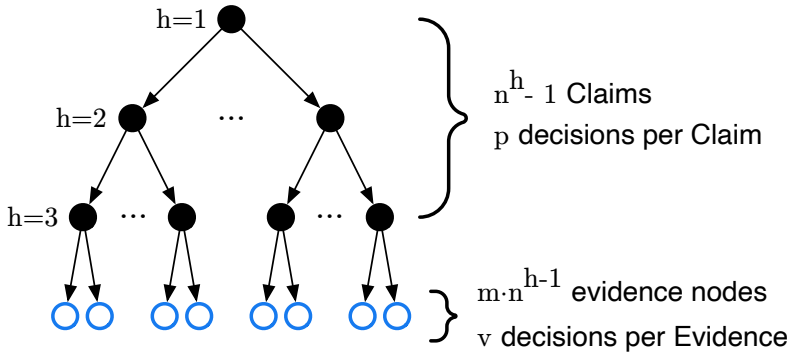


Figure 7.3: Model of AC argument for scalability analysis.

tree has n^{h-1} parent nodes that are each supported by $m \in \mathbb{N}$ evidence nodes. In total there are $m \cdot n^{h-1}$ evidence nodes. Let $v \in \mathbb{N}$ denote the number of decisions that a user must make about each evidence node.

Calculating Worst Case Decision Complexity

Using the above construction, the number of decisions that a user of a CAM must make to apply the method in the *worst* case is:

$$O_d\Big(\big[p \cdot (n^h - 1)\big] + \big[v \cdot m \cdot n^{h-1}\big]\Big) \tag{7.1}$$

Where $O(\ldots)$ is the "Big O" notation from theoretical computer science, the first term is the number of decisions that must be made to configure the belief propagation among the claims, and the second term gives the number of leaf valuations.

Calculating Average Case Decision Complexity

In practice, the user of a quantitative CAM does not need to make a decision for every analysis input. Method implementations or tools can use sensible defaults that represent the most common configurations for belief propagation and assignment. The above decision complexity for the worst case can be generalized to account for scenarios where fewer than the maximum number of decisions is made per parent claim and evidence node.

Let α_n and α_m denote the proportion of cases where a user of a CAM must fully specify an argument step's configuration (p decisions) or leaf valuation (v decisions). In the other $1 - \alpha$ cases, a smaller number of decisions are required, p' and v' respec-

tively. Equation 7.1 is generalized to give the *average* decision complexity:

$$\Omega_d\Big((n^h-1)\cdot\Big[\alpha_n\cdot p+(1-\alpha_n)\cdot p'\Big]+m\cdot n^{h-1}\Big[\alpha_m\cdot v+(1-\alpha_m)\cdot v'\Big]\Big) \qquad (7.2)$$

Observe that when $\alpha_n = \alpha_m = 1$, then the average case decision complexity becomes the worst case complexity from above.

Estimating Effort

Including additional parameters in the above expressions enables users to estimate how long it will take to apply a CAM in terms of minutes or hours of effort. Let t_p denote the time in minutes required to make a decision about a single parameter to configure belief propagation in an argument step; the total time to configure an argument step is $p \cdot t_p$ or $p' \cdot t_p$, respectively. Similarly, let t_v be the time in minutes to make a decision about a single leaf valuation; the total time to evaluate a piece of evidence is $v \cdot t_v$, or $v' \cdot t_v$, respectively. The number of minutes of effort is then computed by including the timing parameters:

$$T = t_p\cdot(n^h-1)\cdot\Big[\alpha_n\cdot p+(1-\alpha_n)\cdot p'\Big]+t_v\cdot m\cdot n^{h-1}\Big[\alpha_m\cdot v+(1-\alpha_m)\cdot v'\Big] \quad (7.3)$$

7.2.2 Results

The worst and average case decision complexities for the *Certus*, BBN method, and DST methods is now compared using the measures above. The results are visualized in Figure 7.4 with the full results in Table 7.5. The parameter choices for the calculations are discussed below.

Parameter Choices for the Argument Model

For a fair comparison among CAMs, the argument is held constant. So, on one hand, the choice of parameters for the argument model (n and m) does not matter. On the other hand, for the analysis to produce useful estimates of decision complexity and effort, selecting parameters that are somewhat representative helps to position the analysis results in terms of day-to-day constraints faced by practitioners. Reports of real-world ACs from the literature are helpful in this regard. Table 7.4 compiles node counts made available in EA case studies. The goal is not to exhaustively enumerate

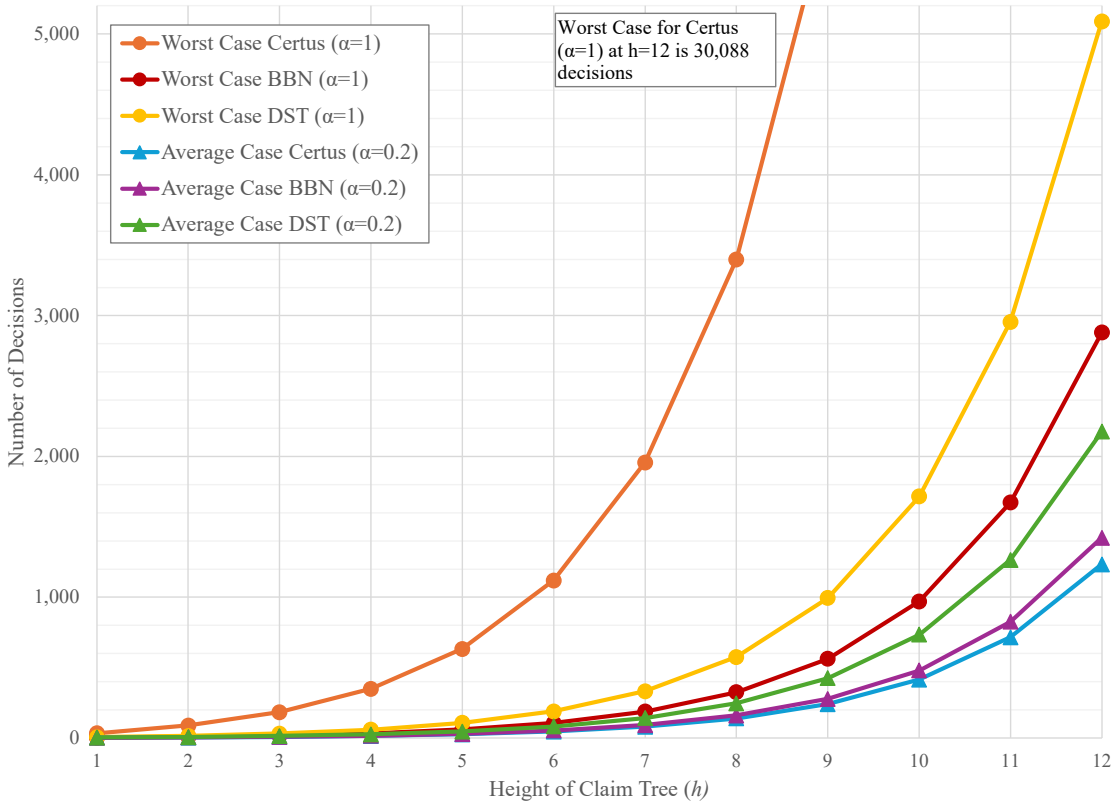


Figure 7.4: Visualization of results from scalability analysis for $1 \leq h \leq 12$.

the argument sizes from all case studies, but rather to find a realistic size for the argument model used in the scalability analysis.

To compute the number of leaf nodes, the number of evidence and residual defeater nodes is summed, and the number of internal arguments is the number of claims and defeaters (minus residuals). Using the averages from Table 7.4, this gives the following pair of constraints:

$$(144 + 99 - 21) = n^h - 1 \quad \text{and} \quad (110 + 21) = m \cdot n^{h-1}$$
$$222 = n^h - 1 \quad \text{and} \quad 131 = m \cdot n^{h-1}$$

There are three unknowns and only two expressions, so exact values cannot be determined. However, taking $h = 10$, allows for n and m to be computed: $n \approx 1.72$ and $m \approx 1$. Interestingly, these values are reasonable for an AC, with each argument step having 1-2 child claims and each line of reasoning ending with one evidence node.

For the average case computation, let $\alpha_n = \alpha_m = 0.20$. This corresponds 1

Argument	Claims	Evidence	Defeaters	Res.	Sum
CERN LHC MPS [91]	146	70	105	9	330
Argument #1 from [53]	184	147	153	19	503
Argument #2 from [53]	188	144	130	28	490
Argument #3 from [53]	94	81	79	7	261
ILI Trust Argument from [209]	142	165	45	-	352
Ventilator Argument from [210]	113	54	81	37	285
Average	144	110	99	21	370

Table 7.4: Reported node counts for a selection of EA arguments in the literature.

in 5 argument steps requiring non-trivial decision-making about how belief should be propagated through the argument. This value for α was chosen based on the results of the expressivity analysis (Section 7.4) and case study (Section 7.5). In both evaluation activities, which considered a number of different argument fragments, 21% of the reasoning steps required custom expressions to model belief propagation[1].

Parameter Choices for the BBN Method

In the worst case decision complexity, a user of the BBN method must make $p = n+2$ decisions per parent claim in the argument. Each child has a link weight parameter and then the combinator and leakage parameter must be selected. For evidence a single decision is made per node, so $v = 1$.

For the average decision complexity, let the α case be the same as the worst case above ($p = n + 2$, $v = 1$). However, suppose that for the $1 - \alpha$ case, tooling assists the user so that they must only select the combinator (NoisyAND or NoisyOR) for each parent claim. Further, suppose the tool also assigns link weights for each child to evenly distribute their contribution to the parent (e.g., $w_i = 1/n$ for NoisyAND). The leakage parameter defaults to $k = 1$, giving $p' = 1$. For evidence, the user is still required to assign a belief for each leaf node, so $v' = 1$.

[1]Originally $\alpha = 0.05$ was chosen. But the analysis was re-run with $\alpha = 0.20$ after completing the expressivity analysis and case study. While the specific decision and time numbers changed, the overall trend between the different CAMs was the same.

Parameter Choices for the DST Method

For the worst case decision complexity, a user of the DST method must make $p = 2n + 3$ decisions per parent claim in the argument. Each child has two parameters (the forward and reverse parameter) and each parent has a combinator, a combining forward, and combining reverse parameter. For evidence, two decisions must be provided per node, one for the "decision" and "confidence", so $v = 2$.

For the average decision complexity, let the α case be the same as the worst case above ($p = 2n + 3$, $v = 2$). However, suppose that for the $1 - \alpha$ case tooling assists the user so that they only need to select a combinator, so $p' = 1$. For the evidence, two decisions must still be made, so $v' = 2$.

Parameter Choices for *Certus*

For the worst case decision complexity, suppose that a user of *Certus* must write a custom belief propagation expression over all child nodes in an argument step. Further, in the worst case, it might be necessary to manually write a `cases` expression where every combination of the canonical sets and child nodes is covered. For an argument step $c_0 \multimap c_1, c_2$, this would be:

```
c0 is cases
    c1 is reject and c2 is reject -> reject,
    c1 is reject and c2 is vopp -> reject,
    c1 is reject and c2 is opp -> vopp,
    ...
    c1 is certain and c2 is vhigh -> vhigh,
    c1 is certain and c2 is certain -> certain,
    otherwise uncertain
```

There are 9 canonical sets defined by *Certus*, so this gives $p = 9^n$ decisions per argument step for the worst case. The number of leaf valuations for evidence is $v = 1$, since a user can make an assignment with a single decision (e.g., `e1234 is low`).

The worst case for *Certus* described above is quite extreme. Several features of the *Certus* language exist to avoid such scenarios, including macros, user-defined operators, and more efficient[2] comparison operators (e.g., `>=`). For the average case, suppose a user must manually craft a belief propagation expression with $p = n$ decisions α percent of the time, and in the other $1 - \alpha$ cases they can use *Certus*' macros or a user-defined operator that require only a single decision, so $p' = 1$. Valuation of leaf nodes remains as $v' = 1$ since every leaf must be assigned a belief level as input.

[2]Efficiency refers to user decisions, not the computational or space efficiency of the algorithm.

Parameter Choices for Time Estimates

Two parameters must be chosen to translate decision complexity into effort estimates. For simplicity, the same parameters are used in both the worst and average case analyses, though different values could be chosen for either case to model different scenarios. Suppose that each evidence valuation takes (on average) $t_v = 5$ minutes of effort to complete. In practice, some might be much faster and others might take significantly longer. For example, many evidence checks amount to simple binary decisions (e.g., "was this document approved?"), which could be completed within a few seconds, but others might require lengthy reviews, consultation with interest holders, and so on. For the argument steps, suppose that each decision takes $t_p = 0.5$ minutes (30 seconds) to complete. As with evidence valuation, this is intended as an average, with many decisions taking just a few seconds and others requiring time.

Table 7.5: Detailed scalability analysis results for the *Certus*, BBN, and DST methods in the worst and average cases. Effort estimates, T, given in hours of effort. Highlighting on row $h = 10$ shows results for a representative argument size. See discussion in Section 7.2.2 for model detailed parameters.

Argument Size				**Worst Case** ($\alpha = 1$)						**Average Case** ($\alpha = 0.20$)					
				Certus		BBN		DST		Certus		BBN		DST	
h	$\|C\|$	$\|E\|$	$\|C\| + \|E\|$	O_d	T [h]	O_d	T [h]	O_d	T [h]	Ω_d	T [h]	Ω_d	T [h]	Ω_d	T [h]
1	1	1	2	34	1	4	0	7	0	2	0	2	0	4	0
2	2	2	4	89	2	9	0	16	0	4	0	5	0	8	0
3	4	3	7	185	5	18	0	32	1	8	1	9	0	14	1
4	8	5	13	350	9	34	1	60	1	15	1	17	1	26	1
5	14	9	23	633	17	61	1	108	2	27	2	30	1	47	2
6	25	15	40	1,120	30	108	2	190	4	47	3	53	2	82	3
7	44	26	69	1,958	52	188	4	332	7	81	5	93	3	143	5
8	76	45	120	3,399	90	326	6	576	11	140	8	161	5	247	9
9	131	77	207	5,878	156	563	10	995	20	241	14	278	8	426	15
10	226	132	357	10,141	270	971	18	1,716	34	416	24	480	14	735	26
11	389	227	615	17,475	465	1,673	31	2,957	59	717	41	827	24	1,265	45
12	669	390	1,059	30,088	800	2,880	53	5,091	101	1,234	71	1,423	41	2,177	77
13	1,152	670	1,823	51,783	1,377	4,956	92	8,760	174	2,123	123	2,449	71	3,746	132
14	1,982	1,153	3,135	89,098	2,369	8,527	158	15,073	299	3,652	211	4,214	122	6,445	227
15	3,410	1,983	5,394	153,279	4,075	14,670	271	25,929	514	6,281	363	7,249	209	11,088	390
16	5,867	3,411	9,278	263,672	7,010	25,235	466	44,603	883	10,805	624	12,469	360	19,072	671
17	10,091	5,868	15,959	453,547	12,057	43,407	802	76,722	1,519	18,585	1,074	21,448	619	32,805	1,154
18	17,357	10,092	27,450	780,133	20,739	74,662	1,379	131,967	2,614	31,968	1,847	36,892	1,064	56,427	1,984

7.2.3 Discussion

From the scalability analysis results it is clear that, in the *worst case*, *Certus* requires a user to make significantly more decisions. This result is not surprising given that *Certus* aims to offer more flexibility in propagation expressions, coming at the cost of higher decision complexity in extreme scenarios. Row $h = 10$ in Table 7.5 corresponds to an argument with 357 claim and evidence nodes. In the worst case, applying *Certus* would require 10,141 decisions which might take 270 hours of effort (between 6 and 7 working weeks). By comparison, the BBN and DST methods require 971 decisions (18 hours) and 1,716 decisions (34 hours).

The worst case for *Certus* is calculated based on the user manually making a decision for all combinations of child nodes and canonical sets, which is not a realistic scenario. Indeed, the *Certus* language contains many features intended to address exactly this situation. These features are accounted for in the average case analysis where *Certus* scales slightly better than the BBN and DST methods. In the average case ($\alpha = 0.20$), using *Certus* on an argument with 357 claims and evidence nodes requires 416 decisions which might take 24 hours of effort. The BBN yields comparable results for the average case. The DST method requires a user to make nearly twice as many decisions.

Limitations

This scalability analysis has several limitations impacting how the results should be interpreted and generalized.

Limitations of the Argument Model. As observed above, the model of the argument used for the analysis is artificial, and was chosen for its simplicity. The purpose of the analysis is to compare scalability *between* CAMs; an extremely realistic argument model is not necessary, provided the model accounts for properties that are relevant to the CAM's decision complexity. For instance, a model that only considered argument steps (claims), without evidence, would not be suitable because the three CAMs differ in the number of decisions required per leaf node. A notable limitation is that real-world arguments are not idealized n-ary trees: an argument might have one very long line of reasoning that is considerably deeper than the average, or it might have a handful of "wide" argument steps with many children while others have only one or two children. This concern was partially mitigated by using published case studies to select parameters n, m, and h for the argument model. Compared to

selecting parameters by intuition, this increases the realism of the results, but only to an extent. Due to this limitation, the results from this model are unlikely to perfectly predict decision complexities or effort estimates for real-world ACs. A future analysis could use statistical simulations to more closely approximate real-world ACs. For the best results, the parameters of the simulation study should be based on structural data (size, node type distributions, etc.) provided by industrial partners.

Limitations of the Decision Parameters. The scalability analysis modelled the decisions made by the user of a CAM with a handful of parameters. These parameters correspond to the *number* of decisions that a user must make, but they do not describe the relative *difficulty* of each decision. For instance, Idmessaoud's DST method requires $v = 2$ decisions for a leaf valuation, but each decision is indicated on an ordinal scale with five options. How does this compare to making a single decision with the BBN method's continuous scale over $[0, 1]$? Perhaps picking a specific value in $[0, 1]$ is harder for an AC developer than selecting from an ordinal scale, or vice versa. For simplicity, the model chosen for the scalability analysis does not consider this difference. Even if it did, the parameter choices would be difficult to justify without extensive empirical data on decision making by AC developers who are using specific CAMs.

Limitations of the Timing Parameters. The effort estimates produced by the model are intended to contextualize the results in terms of a relatable measure: time spent on a task. Without this, it is not clear what it practically means for a CAM to require 10,010 or 365 decisions in the worst and average cases, respectively. However, while it is tempting to do so, these should not be taken as actual estimates of effort required to apply a CAM. In particular, the timing parameters t_v and t_p were selected based on intuition, not empirical data. Moreover, the choice of timing parameters might be context dependent, and be impacted by factors such as the skill of the developer or the specific nature of the argument and evidence. A calibration study would be necessary to produce accurate timing parameters for a given context.

7.2.4 Conclusion

Based on the above analysis results and accounting for limitations, in the worst case, a user applying *Certus* to assess belief in an AC will need to make many more decisions to configure the analysis compared to the BBN and DST methods. However, in the average case, taking advantage of the language's macro and user-defined operator,

Certus users will need to make marginally fewer decisions than when using the BBN method, and about half as many decisions than when using the DST method. In addition to the concrete findings about scalability, this evaluation has also proposed a novel method and model for reasoning about scalability of quantitative CAMs. After additional validation and calibration, the model could be used to support planning of AC confidence assessment activities.

7.3 Sensitivity Analysis

Sensitivity to changes in the belief levels of nodes in the argument is an important property of a CAM. For instance, Graydon and Holloway applied tests to several published quantitative CAMs and found that some were not sensitive to "undermined evidence" (e.g., reducing belief in leaf nodes) [80]. Authors developing quantitative CAMs have begun including sensitivity analyses when evaluating their method(s) [50, 148]. Moreover, several of the requirements from the Dialectic Perspective presented in Chapter 4 capture aspects of sensitivity. For instance, R2-8 says that when the belief in a (credible) defeater is increased, the belief in a parent claim should be decreased.

This section evaluates the sensitivity of *Certus*' macros to changes in the belief of child nodes in an argument step. In doing so, some requirements of the dialectic perspective are also shown to be satisfied by the macros. As with the propagation analysis in Section 7.1 above, a sensitivity analysis is not applicable to user-specified `cases` expressions in *Certus*. One of the goals of this work is to enable flexibility in reasoning. A consequence of providing a high-degree of flexibility is that a user can also write custom propagation expressions (using `cases`) that ignore some nodes in an argument step. Therefore, the sensitivity analysis is not applicable to custom expressions since it would only reveal the sensitivity of a propagation operator as the user intended. On the other hand, *Certus*' macros represent built-in propagation operators that, broadly speaking, should be sensitive to changes in belief levels. Of course, some macros are defined such that they disregard changes in the belief of some nodes in an argument step. For example, the `#MIN` macro might not be sensitive to changes in the belief for a premise-typed child if another child is present that always has a lower level of belief.

7.3.1 Method

Some authors have used "tornado plots" to perform sensitivity analyses of CAMs [148, 50]. Tornado plots visualize the effect of extreme belief levels, and work well for methods whose propagation operators are continuous functions. However, *Certus*' macros operate as discrete functions over the language's canonical fuzzy sets and so tornado plots are less informative. Moreover, to accommodate dialectic reasoning, the macros encode functions that cover additional cases that depend on the relative belief levels among child nodes (e.g., Equation 5.13 for the strict `#FUSE` macro). So a multi-dimensional analysis is required.

Two prototypical argument steps with varying numbers of children were used: one with a premise-typed parent and another with a defeater-typed parent. For each of *Certus*' nine macros, a parameter sweep over all possible belief assignments for the children was performed. The set of results was queried to determine if the belief level computed by a macro is sensitive to a change in the children. The results are visualized as two-dimensional heatmaps that are generated by holding belief for two children constant while the other two are varied. The sensitivity of the macros in four cases were investigated:

1. **Change to a Premise's Belief** - In the absence of defeater-typed siblings, when the belief in a premise-typed node changes, the belief in the parent should also change. This covers Graydon and Holloway's "undermined evidence" scenario for situations where the belief in a child decreases, if the premises are taken as evidence nodes [80]. For a defeater-typed parent, this case addresses requirements R2-11 and R2-10. This case is investigated using two argument steps: $c_0 \multimap c_1, c_2$ and $d_0 \multimap c_1, c_2$.

2. **Adding a Defeater** - When a credible defeater is added to an argument step with a premise-typed parent, then requirement R2-1 says the belief in the parent should decrease. When the parent is a defeater, then the added child defeater supports the parent, whose belief should increase. Two argument steps are used: $c_0 \multimap c_1, c_2, d_3$ and $d_0 \multimap c_1, c_2, d_3$.

3. **Change to a Defeater's Belief** - When belief in a child defeater changes, then the belief in a parent claim should change in the opposite direction, per R2-8 and R2-9. The same three-child argument steps as in the previous case are used.

4. **Resolving Defeaters** - When a defeater is resolved, then per R2-4 and R2-5, the belief in the parent should increase (or stay the same). An argument step with three nodes, as above, is compared to an argument step containing an additional defeater: $c_0 \multimap c_1, c_2, d_3, d_4$.

The parameter sweep was performed using the implementation of the *Certus* language provided by the `certus-ts` library described in Section 5.8. In addition to analyzing the sensitivity of the macros, this analysis provided additional verification of the implementation.

7.3.2 Results

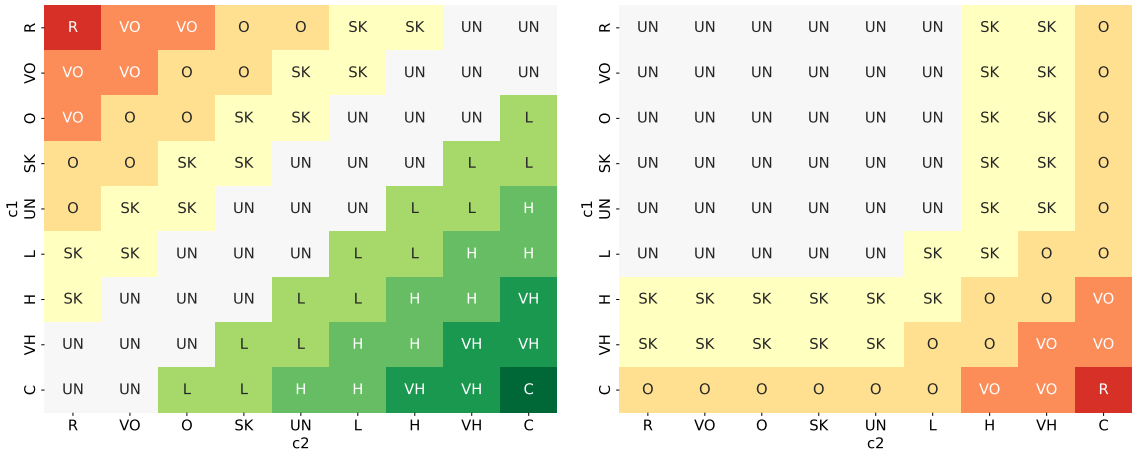
For brevity, only the results for the deductive `#FUSE` macro are presented in detail. The results are organized according to the analysis cases described above. A summary of results for all macros is shown in Table 7.6 with detailed results in Appendix C.

Changes to a Premise's Belief for `#FUSE`

Figure 7.5 shows the response of the deductive `#FUSE` macro to varying the belief in child premises, c_1 and c_2. The case with only premise-typed children is in 7.5a. From this heatmap it is apparent that the macro implements an average among the premise-typed children. Moreover, it can be seen that Graydon and Holloway's undermining requirement is satisfied: when belief in one child premise decreases, then belief in the parent also decreases, as wanted. The case with premise-typed children rebutting a parent defeater is shown in 7.5b. From this plot it can be seen that uncertain (and lower) belief levels for the premises do not rebut a parent defeater; however, once there is positive belief in the premises, the belief in the parent defeater begins to decrease as wanted by R2-10 (and vice versa for R2-11).

Adding a Defeater for `#FUSE`

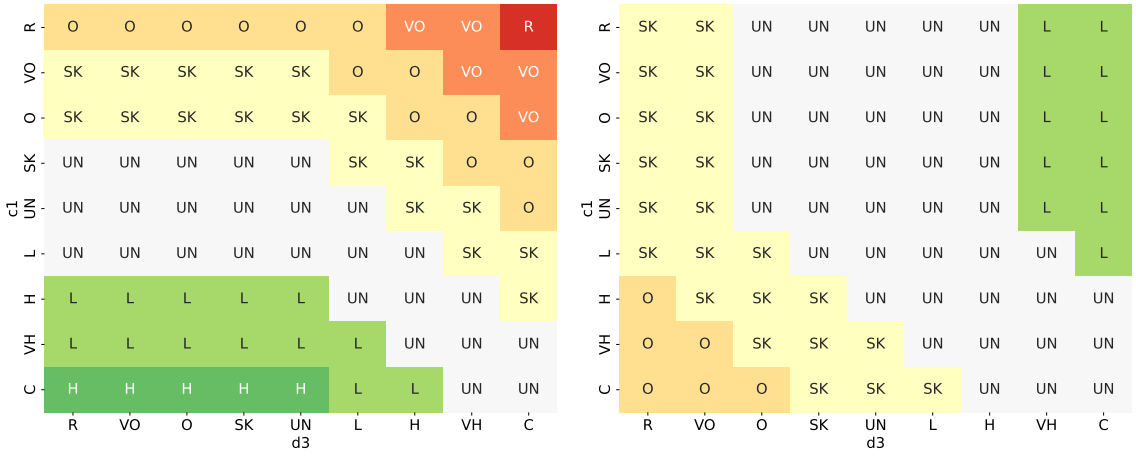
Figure 7.6 shows the response of the deductive `#FUSE` macro to adding a child defeater, d_3. In 7.6a, a defeater is added below a premise-typed parent that already has two child premises. The belief in one child premise, c_1, varied while the belief in the other, c_2 is held at `UN`. As belief in d_3 increases above uncertainty, the belief in the parent premise declines, as wanted by R2-1. Note that when the defeater is not credible (belief `UN` or lower) the belief in the parent is determined by the premise-typed



(a) Step c_0 —● c_1, c_2. (b) Step d_0 —● c_1, c_2.

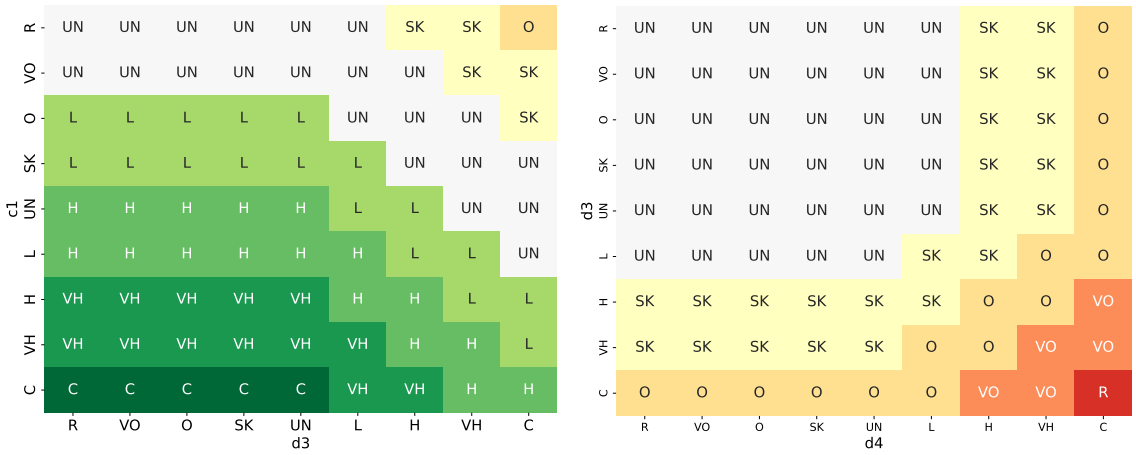
Figure 7.5: Response of the deductive #FUSE macro to changes in belief of children.

children. This behaviour is consistent with R2-1's exception for adding a defeater that is not credible, which is interpreted by the deductive #FUSE macro as anything with belief less than uncertainty. Figure 7.6b shows the case where a new defeater, d_3, is added under a defeater-typed parent. In this case, d_3 provides support for the parent, as can be seen by reading across the rows of the heatmap. As belief in d_3 increases, the belief in the parent also increases.



(a) Step c_0 —● c_1, c_2, d_3; $c_2 =$ UN. (b) Step d_0 —● c_1, c_2, d_3; $c_2 =$ UN.

Figure 7.6: Response of the deductive #FUSE macro to adding a defeater.



(a) Step c_0 —● c_1, c_2, d_3; $c_2 =$ `C`. (b) Step c_0 —● c_1, c_2, d_3, d_4; $c_1 = c_2 =$ `UN`.

Figure 7.7: Response of the deductive `#FUSE` macro to resolving defeaters.

Changes to a Defeater's Belief for `#FUSE`

The results in Figure 7.6a also provide evidence that the deductive `#FUSE` macro satisfies requirements R2-8 and R2-9. For R2-8, as the belief in d_3 increases the belief in the parent premise declines, and vice-versa for R2-9.

Resolving Defeaters for `#FUSE`

Figure 7.7 shows the response of the deductive `#FUSE` macro when a defeater challenging a parent premise is resolved. In 7.7a, the situation where belief in a single child defeater, d_3, declines is considered. Per R2-4, as the belief decreases, the belief in the parent increases. Note that in the `#FUSE` macro's definition, once the defeater becomes uncertain (`UN`), it no longer reduces belief in the parent, and is considered to be "resolved". This choice was discussed in Section 5.5.2 as part of formulating the deductive heuristic underpinning the macro; other macros take a more aggressive stance towards resolution. Figure 7.7b considers the situation where multiple defeaters, d_3 and d_4, are resolved while the supporting premises are held at uncertain belief. When both d_3 and d_4 are rejected, the belief in the premise-typed parent returns to uncertainty in accordance with R2-5 under the deductive interpretation.

Summary of Results for Remaining Macros

The analysis cases applied above to the deductive `#FUSE` macro were repeated for the remaining eight *Certus* macros. The results are summarized in Table 7.6 and are as

wanted for each analysis case, up to the constraints of each macro's definition. For instance, the #MIN macros were not sensitive to changes in the belief of child nodes if those changes were not the minimum belief level among the children. This is the expected and desired behaviour for these macros. When the minimum value was changed, then the macro exhibits the desired sensitivity.

Table 7.6: Summary of sensitivity analysis results for *Certus* macros.

	Deductive			**Eliminative**			**Strict**		
Analysis Case	FUSE[1]	MIN[1,2]	MAX[1,2]	FUSE	MIN[2]	MAX[2]	FUSE[3]	MIN[2,3]	MAX[2,3]
Undermined Evidence	✓	✓	✓	✓	✓	✓	✓	✓	✓
Claim Increase (R2-10)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Claim Decrease (R2-11)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Adding a Defeater (R2-1)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Defeater Increase (R2-8)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Defeater Decrease (R2-9)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Resolve One Defeater (R2-4)	✓	✓	✓	✓	✓	✓	✓	✓	✓
Resolve All Defeaters (R2-5)	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] Up to uncertainty (UN), per deductive heuristic.
[2] Up to the minimum or maximum value among the children, respectively.
[3] Up to the strict interpretation of defeaters.

7.3.3 Discussion

Based on the results of the sensitivity analysis presented above, it can be seen that *Certus*' macros are sensitive to changes in their inputs, at least for the selected analysis cases over the canonical sets. Sensitivity is a desirable property for quantitative CAMs, since it ensures that changes to the belief at the leaves of an AC's argument are able to impact the belief computed for the top-level claim. Additionally, compared to previous sensitivity analyses performed for other CAMs [50, 148], this analysis considered sensitivity to the credibility of defeaters. These results provided evidence that several of the requirements from the Dialectic Perspective developed in Chapter 4 are satisfied by *Certus*' macros. Finally, the analysis served as a verification activity, confirming that the implementation of the macros in `certus-ts` is consistent with the intent of the macros expressed in Chapter 5.

Limitations

This sensitivity analysis has several limitations that impact how the results should be interpreted.

Limitations of Discrete Values. Often sensitivity analyses are performed for continuous functions. Though the belief scale used by *Certus* is on the continuous interval $[-1, 1]$, the macros operate over *Certus*' canonical sets (e.g., `C`, `VH`, ... `R`) which are treated as discrete values over this interval. This analysis made the simplifying assumption that the behaviour of the macros can be analyzed by varying the inputs over the canonical sets, and that a higher resolution analysis on $[-1, 1]$ is not required. This assumption is justified by considering the definitions of the macros, which use various operators to find the closest canonical set for each child node's belief. When a user represents their belief in the argument using only the canonical sets, as is intended for the majority of the language's use cases, this assumption will hold. However, if belief is modelled with user-defined sets that do not align with the canonical sets, then the macros might not be sensitive to certain changes in belief.

Limitations in the Cases Analyzed. Four cases, corresponding to eight requirements, were analyzed in detail for each of *Certus*' macros. However, these cases represent only a handful of potential argument step configurations. Moreover, it is difficult to visualize the results of the analysis in higher than two dimensions. It remains possible that some scenarios exist where the macros do not behave as wanted. However, this is unlikely for at least two reasons. First, within the scenarios considered, the macros exhibited well-defined behaviour that is predicted by their formal definition(s) and that is consistent with their underlying intent. Second, the cases analyzed cover a range of representative argument steps, including those with premise- and defeater-typed parents and varying numbers and types of child nodes. Future work might seek to remove this limitation by preparing formal proofs to show that the macros possess desirable properties.

Limitations for Argument Depth. This analysis only considered arguments with a single reasoning step. It is possible that the sensitivity properties determined for a single argument step will not scale over the multiple reasoning steps that commonly occur in real-world ACs. This could result in a situation where the impact of a leaf node's belief is not "felt" at the top-level node. This situation is different than the attenuation and amplification effects analyzed in Section 7.1, where the belief changed

due to effects of multiple numerical operations stacking on top of each other. In fact, in *Certus* it might be intentional that the belief in a leaf node does not propagate, e.g., the `#MIN` macro only propagates the most pessimistic belief. So, while some macros might result in reduced sensitivity over multiple reasoning steps, and these cases were not analyzed, this is not necessarily a problem and in many cases might be desirable behaviour.

7.3.4 Conclusion

Considering the results and limitations of this sensitivity analysis, provided the canonical sets are used to represent belief, it can be concluded that *Certus*' macros are appropriately sensitive to changes in the belief in children for a representative set of argument steps. It is likely that these results can be generalized to other configurations, but this has not been proven by this analysis. Additionally, it is expected that the macros will have similar behaviour if the user-defined sets are similar to *Certus*' canonical sets, but their behaviour is not well understood as the inputs increasingly differ from that of the canonical sets.

7.4 Expressivity Analysis

When applying a quantitative CAM, it is important that the method permit users to express nuanced reasoning when modelling how belief propagates through an argument. The need for expressivity was captured by requirement R6 in Chapter 4. *Certus* is designed to enable a high degree of flexibility in how users model belief propagation, especially when the custom `cases` expression is used. For instance, with the `cases` expression it is possible to adjust the *relevance* of individual children to the parent's belief. This section evaluates the expressivity of *Certus* across a range of argument fragments drawn from published AC case studies.

7.4.1 Method

To evaluate expressivity, *Certus* was applied to a sample of 14 argument fragments from publicly available AC arguments. The argument fragments are summarized in Table 7.7. In their work on using generative AI to create ACs, Odu et al. identified a set of five ACs from the literature that they used for their own evaluation [108]. To

mitigate selection bias, where argument fragments were chosen because they would be "easy" to assess with *Certus*, Odu et al.'s selection was used as the basis for this evaluation. While the selection offers breadth and diversity, the fragments are limited in their size and scope due to publication constraints. So, six additional fragments were selected (hand-picked) from the CERN Large Hadron Collider's (LHC's) large-scale AC [91].

The selection of argument fragments from the CERN AC was guided by the following three criteria. First, since argument structures tend to be different between the top and bottom levels of an AC, to improve diversity, fragments were chosen from near the top of the argument (i.e., without leaf nodes) and also from near the bottom (i.e., containing leaf nodes). Second, since the CERN argument was created using EA, four of the fragments were chosen because they contained defeaters. Finally, fragments must have had at least 5 nodes, though larger fragments were preferred.

Table 7.7: Summary of argument fragments used for the expressivity analysis.

			Node Counts					
Argument	**Domain**	**Attribute**	C	E	S	IR	D	Total
ACAS-Xu [211]	Aviation	Security	10	2	4	0	0	16
BLUEROV [212]	Subsea Robotics	Safety	12	1	3	0	0	16
CAA-1 [213]	Aviation	Safety	5	2	2	0	0	9
CAA-2 [213]	Aviation	Safety	4	0	1	0	0	5
CAA-3 [213]	Aviation	Safety	6	6	2	0	0	14
DeepMind [214]	Health	Safety	15	0	2	0	0	17
GPCA [215]	Health	Safety	14	2	7	0	0	23
IM-Server [216]	Communications	Security	10	0	4	0	0	14
CERN-1 [91]	Nuclear Physics	Safety	5	2	1	1	6	15
CERN-2 [91]	Nuclear Physics	Safety	3	0	1	1	0	5
CERN-3 [91]	Nuclear Physics	Safety	2	1	1	0	2	6
CERN-4 [91]	Nuclear Physics	Safety	5	0	1	1	0	7
CERN-5 [91]	Nuclear Physics	Safety	8	5	1	1	6	21
CERN-6 [91]	Nuclear Physics	Safety	5	4	1	0	3	13

Evaluation was focused on the expressivity of *Certus* for describing belief propagation in argument steps, not belief valuations at the leaves of the argument. Expressing belief valuations at the argument's leaves is not particularly demanding from an

expressivity perspective. Setting aside artifacts and indicators, which were not considered in this analysis, leaf valuation amounts to selecting a canonical set for each leaf node, or in the worst case, supplying a user-defined fuzzy set. On the other hand, there is considerably more variability in belief propagation operations.

For each argument fragment, *Certus* was applied in three steps:

1. **Assign Leaf Valuations** - For each leaf node in the argument fragment, assign a belief valuation from *Certus*' canonical sets. Since the argument fragments are evaluated outside of their project context, the specific value assigned will not be representative of the actual belief in the leaf node. This is acceptable given the focus of the analysis on propagation operators. Valuations were supplied based on a cursory judgement of the leaf node's assertion. For instance, if an evidence leaf asserted "All Tests Passed", then this was taken at face value and assigned *certain* (`C`)[3]. Less precise or factual assertions were assigned lower belief levels while giving the benefit of the doubt to the argument's authors.

2. **Compose Propagation Operations** - For each parent node, a propagation operation is specified. A choice was made between three options: 1) remaining with the default operation (*Certus*' deductive `#FUSE` macro), 2) assigning one of *Certus*' other macros, or 3) composing a custom propagation expression.

3. **Check Output** - Check the computed belief assignments to ensure they align with my intuition about how belief should propagate through the argument for each argument step.

The evaluation was performed using the implementation of `certus-ts` in *Socrates* as described in Section 5.8. Arguments were exported from *Socrates* for subsequent analysis in a Python notebook (e.g., to tabulate the number of macro uses).

7.4.2 Results

Overall, *Certus* was able to express propagation operators for all argument steps in the selection of argument fragments. The full set of argument fragments used for the evaluation, annotated with *Certus* expressions, are provided in Appendix D.

[3]We might be *certain* (up to tool qualification for the test infrastructure) of the fact that all tests have passed. However, this does not mean that claims supported by this assertion about the quality of testing will also be *certain*. Other factors such as the quality of the test cases must also be considered.

As an example, the CAA-1 argument fragment is shown in Figure 7.8. The argument focuses on the ability of a collision avoidance system to provide correct instructions to Unmanned Aerial Vehicles (UAVs). Belief valuations are shown as annotations on the nodes, with solid borders indicating leaf assignments and dashed borders indicating computed values. *Certus* propagation operations that were specified as part of the evaluation are shown, default propagation operators are omitted for brevity. Non-leaf nodes without a propagation operator used the default operator (`#FUSE`). Some of the observations from the expressivity analysis make reference to the propagation operations from this fragment.

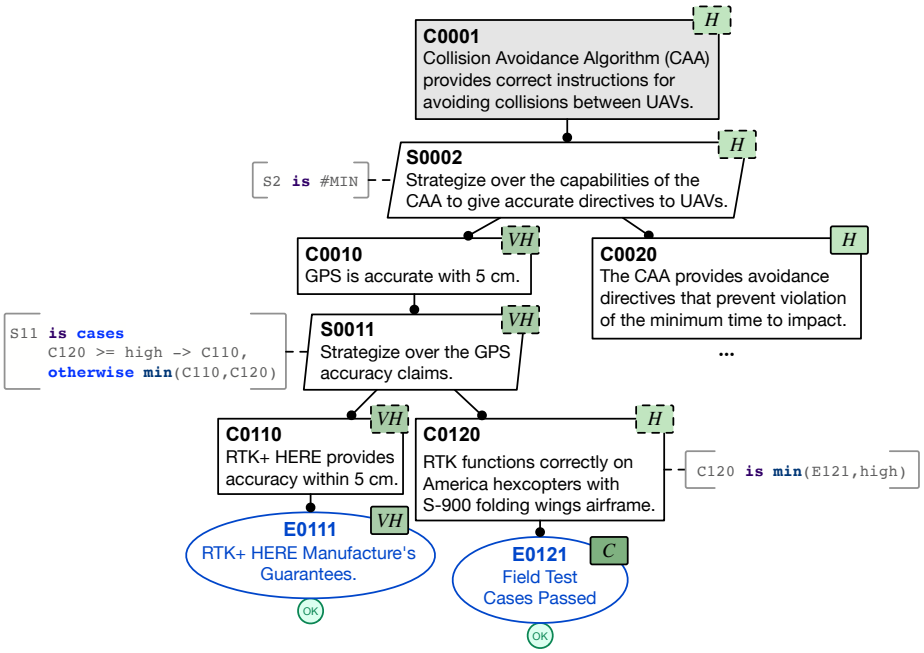


Figure 7.8: The CAA-1 argument fragment (from [213]) from the expressivity analysis.

Observation: Belief Bounding

A common pattern among the custom propagation operations was bounding the belief propagating from one or more children to a parent. An example appears in the propagation operation annotating `C120` in Figure 7.8 where belief that the RTK GPS sensor functions correctly is bounded to at most `high`, even if a higher valuation is provided for the supporting evidence. Field test case results are presented as evidence in `E121`. However, no argument is provided about the quality of the tests (completeness, correctness, etc.). Additionally, field-testing alone is not sufficient to

show that the integration of a sensor into a larger vehicle platform is acceptable; design rationale and bench tests might also be required.

The concept of belief propagation bounding is not unique to *Certus*. Hobbs and Lloyd's BBN method includes the notion of "leakage" which modifies the propagated belief to account for incompleteness in an argument step, and Idmessaoud's DST method has forward and reverse parameters for a similar purpose.

Observation: Preference for `MIN` Over `MAX`

There was a preference to use the `#MIN` macro over the `#MAX` as a propagation operator. For example, node `S2` in Figure 7.8 makes use of the deductive `#MIN` macro to propagate the belief from `C10` and `C20`. Since both of these claims must be true for the top-level claim, `C1`, to be true, the minimum belief among the children is selected. This is analogous to a logical "and" operation. Table 7.8 shows that the `#MIN` macro was used 14 times in the analysis while the `#MAX` macro, which corresponds to logical "or", was not used at all. This suggests that, at least for the fragments analyzed, there is a preference by the AC's authors to provide reasoning steps where all premises are true, which is readily modelled with `#MIN`. However, this is not universally the case, since one can imagine arguments corresponding to systems with redundant channels or certification activities where alternative compliance requirements are satisfied, both of which could be modelled with `#MAX`. Nonetheless, the fact that `#MAX` was not used at all in this analysis is interesting.

Observation: Balancing Belief with `#FUSE`

Though not shown in the fragment in Figure 7.8, the `#FUSE` macro was also explicitly assigned 14 times across the analysis. These corresponded to situations where belief in the parent is the result of balancing child beliefs. For example, in the ACAS-Xu fragment, `#FUSE` is used to merge belief from child claims corresponding to different phases of the system development lifecycle: a lower belief in one phase could be made up for by higher belief in another phase. Additionally, since `#FUSE` is the default, it was also used numerous times; though these were mainly singleton argument steps, consisting of a parent supported by a single child (e.g., `C1` —• `S2`).

Observation: Conditional Belief Propagation

A common pattern applied during the analysis was *conditional belief propagation*, where the belief in the parent rests on one (or more) children, but only if belief in another child achieves a threshold level. This pattern is used on strategy `S11` in Figure 7.8, where belief in the accuracy of the GPS depends on the manufacturer's specification for the sensor (`C110`), but only if there is `high` belief that the sensor is correctly installed on the UAV (`C120`). If this threshold for correct installation is not satisfied, then the minimum belief among the children is used. This pattern was also discussed in Sections 5.1 and 5.2.

The same pattern was used for belief propagation in some argument steps that included an inference rule (`IR`). Belief propagation was conditioned on the level of belief in the inference rule achieving a minimum threshold. Given an argument step `C0` —• `C1,C2,IR3`, the propagation operation would be:

```
C0 is cases
    IR3 >= high -> min(C1, C2),
    otherwise min(C1, C2, C3)
```

Observation: Relevance of Children to a Parent Node

Sometimes the belief in a parent node depends on several children, but each child's belief does not contribute uniformly to the parent. That is, some children might be more *relevant* to their parent than others. The BBN- and DST-based methods model relevance using numerical weighting parameters (e.g., the BBN method's link probability parameter). In *Certus*, relevance can be expressed in at least two ways. First, a `cases` expression can be composed that covers combinations of child beliefs and encodes a weighting function over the children. This works for small argument steps, but can create large numbers of conditions as the number of children grows.

The second approach also uses *Certus*' `cases` expression, but applies a similar strategy to the one discussed above for conditional belief propagation. Given an argument step `C0` —• `C1,C2,C3`, suppose that child `C1` is more relevant to the parent than the other children such that `C0`'s belief should rest mainly on `C1`. If `C2` or `C3` have high enough belief, then they may also contribute to the parent. The following expression encodes this relevance relationship, where `C1`'s belief makes up a substantial portion of the belief in the parent (up to `high`), and then the remainder is composed of the additional belief contributed from `C2` and `C3`.

```
C0 is cases
    C1 >= high -> max(high, min(C2, C3)),
    otherwise C1
```

This second pattern was used in the GPCA fragment (see Figure D.7) to model different levels of relevance of among several sub-claims, one of which was about the mitigation of system hazards that are the most relevant to the subject of the fragment's top-level claim, which was about system safety.

Summary of Results

A summary of results for the expressivity analysis is shown in Table 7.8, which tabulates the number of uses for each propagation operator, including custom expressions and the default operator. Overall, the default propagation operator was the most common and was used in 59 argument steps, which corresponds to 54% of all argument steps across the sampled arguments. *Certus*' macros were used in 28 argument steps (26%), with the most common macro being the deductive `#MIN` macro. Custom propagation expressions were composed for 23 argument steps (21%).

The eliminative macros were only used in the CERN argument fragments. The CERN case study explicitly used the EA method [91], and so the argument steps that involved defeaters were modelled with the eliminative interpretation. No macros from the strict interpretation were used, though it is worth noting that in the absence of defeaters, the deductive, eliminative, and strict variants of the macros are the same.

7.4.3 Discussion

From the analysis results and observations presented above, it is apparent that *Certus* is expressive enough to model belief propagation in a range of argument steps. Among *Certus*' pre-defined macros, the `#MIN` and `#FUSE` macros are widely applicable, though they are not sufficient to capture all situations. Custom propagation expressions must be crafted to express concepts like bounded or condition belief propagation and relevance. The results of this analysis suggest that additional macros (or other built-in operators) could be defined that capture this type of reasoning.

Limitations

The results of this expressivity analysis are subject to several limitations that impact how the results should be interpreted.

Table 7.8: Summary of expressivity results for *Certus* for the analyzed argument fragments. Includes macros, custom, and default ("Def.") propagation operations.

	Structure		**Deductive**			**Eliminative**			**Strict**				
Argument	Steps	Nodes	FUSE	MIN	MAX	FUSE	MIN	MAX	FUSE	MIN	MAX	Cust.	Def.
ACAS Xu	10	16	0	1	0	0	0	0	0	0	0	2	7
BlueRov	8	16	1	0	0	0	0	0	0	0	0	2	5
CAA 1	6	9	0	1	0	0	0	0	0	0	0	2	3
CAA 2	2	5	0	1	0	0	0	0	0	0	0	0	1
CAA 3	8	14	0	1	0	0	0	0	0	0	0	4	3
DeepMind	11	17	0	3	0	0	0	0	0	0	0	0	8
GPCA	12	23	0	2	0	0	0	0	0	0	0	3	7
IM Server	8	14	0	1	0	0	0	0	0	0	0	0	7
CERN 1	12	15	0	1	0	5	0	0	0	0	0	0	6
CERN 2	2	5	0	0	0	0	0	0	0	0	0	1	1
CERN 3	5	6	0	0	0	1	0	0	0	0	0	1	3
CERN 4	2	7	0	1	0	0	0	0	0	0	0	0	1
CERN 5	15	21	1	0	0	4	1	0	0	0	0	4	5
CERN 6	9	13	0	1	0	2	0	0	0	0	0	4	2
Total	110	181	2	13	0	12	1	0	0	0	0	23	59
Percent	-	-	2%	12%	0%	11%	1%	0%	0%	0%	0%	21%	54%

Limitation: Argument Sample Size and Quality. The argument fragments selected for this analysis might not be representative of all AC arguments. The space of potential assurance arguments that could be made for critical systems is large. Moreover, though structured notations constrain the syntax of potential argument structures, there is significant variability in terms of content (i.e., what claims are made). I am not aware of any studies that attempt to catalogue all types of arguments that could be used as a basis for a definitive expressivity analysis, and it is not obvious how such a catalogue could be credibly composed. As a result, it is possible that the findings of the analysis will not generalize to all types of AC arguments. As a partial mitigation for this, the analyzed argument fragments have considerable diversity in terms of their authorship, style, domain, and relevant quality attributes. Future work might further reduce this limitation by surveying a wider range of AC arguments to extract fragments to identify reasoning patterns.

Limitation: Confirmation Bias. Only one person (myself) undertook the expressivity analysis. As a result, it is possible that results were biased towards showing *Certus*' expressivity. There are two places in the evaluation protocol this could have manifested. First, the selected argument fragments could have been preferentially selected to advantage *Certus*. This was mitigated by using arguments selected by another group of authors [108], and then supplementing the sample with argument fragments that were hand-picked from a real-world case study. Second, the selection of propagation operators might have been biased towards confirming *Certus*' expressivity. To some extent, this type of bias is not relevant, since the objective of this evaluation is to determine if it is *possible* for *Certus* to express belief propagation operations. The evaluation procedure did not involve judgement about whether *Certus* is more or less expressive than an alternative. To the extent this bias is relevant, it might have manifested as trying to unreasonably manipulate *Certus*' expressions to fit the reasoning in the argument fragments[4]. My intuition is that this did not happen, and that *Certus* was able to express belief propagation without undue effort or manipulation. The detailed results are available in Appendix D for inspection. It is notable that the custom propagation operators used are not very complex.

7.4.4 Conclusion

Considering the results and limitations of this analysis, it can be concluded that *Certus* is capable of expressing belief propagation operations for a reasonably wide range of AC arguments. However, because the analysis did not exhaustively examine all possible argument variants, this claim cannot be generalized to all arguments. Future work might aim to further validate expressivity on a larger corpus of argument fragments so that this claim can be generalized further. Additionally, it might be interesting to compare the expressivity of *Certus* against that of other quantitative CAMs, but a more rigorous evaluation procedure would be required to mitigate bias.

[4] "A square peg into a round hole"

7.5 Case Study: Continuous Assurance of a Rapidly Manufactured Medical Ventilator

The evaluation activities described above test for specific properties of the *Certus* language in isolation. It is also necessary to show that the *Certus* language and method produce acceptable results when applied to a realistic DAC. This section applies *Certus* to assess belief in an exemplar DAC that was developed for the software controlling a medical ventilator system called *gVent* (Gravity Ventilator).

The *gVent* system was developed by a group[5] of researchers, engineers, and medical practitioners at the University of British Columbia in Vancouver, Canada in response to sudden demand for medical ventilators during the global COVID-19 pandemic. Their goal was to develop a low-cost rapidly manufactured ventilator system (RMVS) that could be assembled from widely available components [217]. Unlike other RMVS solutions developed during the COVID-19 pandemic, *gVent* used gravity to pressurize air for ventilation, rather than relying on air compressors or pumps, which require additional components and maintenance. The project produced a prototype to prove the system concept before it was discontinued in 2022. Resources from the project are available under an open-source license on GitHub[6]. An overview of the *gVent* system is shown in Figure 7.9.

Critical Systems Labs (CSL) was engaged as part of the *gVent* project to provide guidance on system design and safety assurance. After the project concluded, CSL developed a publicly available safety assurance case for *gVent*'s control software, which is now available on the company's website[7]. The objective of this exemplar was to demonstrate the use of EA on a representative medical device. CSL's initial AC was not developed as a DAC as contemplated by Definition 5. So, the present case study will build upon CSL's AC by introducing several dynamic aspects that could conceivably be incorporated into a process of continuous assurance for *gVent*.

The objective of this exemplary case study is to provide additional evidence that *Certus* is a trustworthy CAM. More concretely, the aim is to answer the following research questions:

1. **Applicability** - Is *Certus applicable* to a real-world DAC, including during the

[5] `https://www.cosmicmedical.ca/`
[6] `https://github.com/COSMIC-medical/gVent`
[7] `https://criticalsystemslabs.com/perspective/`

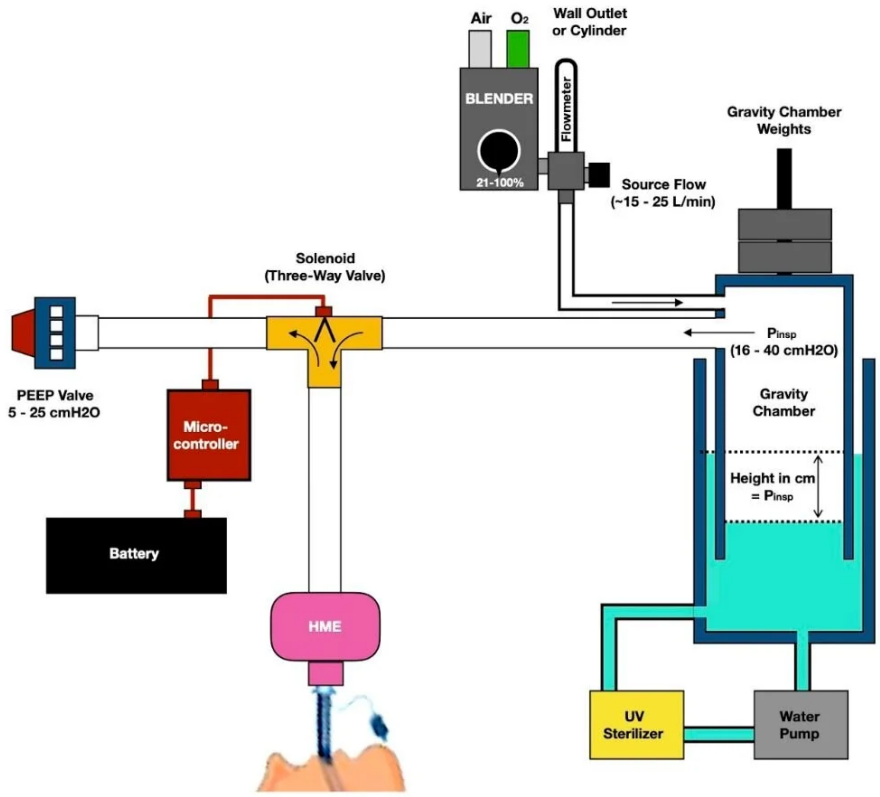


Figure 7.9: Overview of the *gVent* system used in the case study.

deployment and evolution of the system?

2. **Sensitivity** - When used on a DAC with a larger argument, does *Certus* produce belief assessments that are *sensitive* to belief valuations driven by dynamic aspects such as artifacts and indicators?

3. **Expressivity** - Can the *Certus* language express belief propagation operations that are present in real-world DACs?

4. **Effort** - How much effort is required to apply *Certus* to a real-world DAC?

Disclosure. I am currently employed by Critical Systems Labs Inc. I contributed to the company's initial engagement with the *gVent* project, including co-authoring drafts of the software requirements specification and software architecture documents. However, I was not involved in CSL's development of the AC case study aside from providing generic guidance on AC notation and tooling to the case's authors.

7.5.1 Case Study Narrative

This case study positions the development of the *gVent* system within a process of continuous assurance where data collected from field monitoring is used to make decisions about operations and improvements. A scenario is imagined involving two iterations of the *Certus* Method where the *gVent* AC is used as a decision-making aid during system development, deployment, and evolution:

1. The *gVent* system has matured to the point where it is ready for human use as part of the response to the COVID-19 pandemic. The AC has been created and *Certus* is applied as part of deployment decision-making.

2. *Certus* produces a valuation that exceeds the deployment threshold of *high*[8] belief for the argument's top-level claim.

3. Sometime after deployment, data from *gVent*'s field monitoring program becomes available indicating that the control software is failing due to out-of-memory errors. This triggers *Certus* to produce a belief valuation for the top-level claim that is below the operating threshold.

4. Clinical users are notified of a suspected software defect. A recommendation is provided to either stop using *gVent* until the software is updated or to ensure the device is restarted once every 24 hours to reset the memory usage[9].

5. The *gVent* team fixes the memory issue and produce a new release of the software. They update the AC with the additional argumentation to reflect the changes to the software and support the sub-argument with unit test evidence. They opt not to repeat some of the more laborious verification activities (e.g., system-level testing); these artifacts are not linked to the updated AC.

6. When *Certus* is used to re-compute belief in the argument as part of the deployment decision-making process, the belief valuation is below the deployment threshold due to the missing evidence. They cannot release the system.

7. The *gVent* team completes the necessary verification activities and re-runs *Certus*. They are now able to (re-)deploy the system.

[8]One could argue that the decision threshold should be *very high* even *certain*. However, *gVent* is intended for use in exceptional circumstances (e.g., a global pandemic) which justifies a lower deployment threshold. This threshold is illustrative and might not be acceptable for a real RMVS.

[9]In the context of a global pandemic, providing a short-term "work around" is viewed as a practical and necessary solution when the alternative might be not providing ventilation for patients.

8. After deployment, they continue to monitor indicators. There are some reports of over-voltage events. Providing this data to the AC (via *Certus*) results in the operational threshold being violated. Again, a notice is sent to clinicians to stop using *gVent* until the issue is addressed.

9. *... and the process of continuous improvement and assurance continues.*

7.5.2 Method

The methodology for the case study is adapted[10] from the guidance on case study research from Host et al. [218]. The case study was performed in the *Socrates* tool using the implementation of *Certus* described in Chapter 5.

Case Selection

There are many examples of AC arguments published in the literature. However, not all of these are applicable for the research questions listed above. The following criteria were used to select the AC for this case study:

1. **Realism** - The AC should have been developed for a real-world critical system. High-fidelity simulations or prototype systems are acceptable, provided they are sophisticated enough to generate evidence for an AC.

2. **Availability** - The complete AC argument must be publicly available.

3. **Structured Argument** - The AC's argument must be expressed with a structured notation (e.g., GSN, CAE, EA) and contain at least 100 structural nodes.

4. **Dialectic Reasoning** - The AC's argument must use dialectic reasoning.

5. **Independence** - I must not have been involved in the creation of the AC, or at least have been at "arms-length" from its development.

6. **Dynamic Aspects** - The AC should include dynamic aspects, such as indicators or dynamic artifact linking. If this criterion is not satisfied, it might be necessary to modify the selected AC to include this type of information.

[10] It is worth noting that some aspects of Host et al.'s research method are not applicable to the present case study since they focused on studies where data is gathered *in situ*. Nonetheless, Host et al.'s guidance was valuable in shaping this study's methodology.

No publicly available ACs were found that satisfy all the above criteria. The CERN Large Hadron Collider's argument satisfies all but the independence and dynamic criteria: I was involved in preparation of this case [91, 174], and it does not have dynamic aspects. The *gVent* AC satisfies all but the dynamic criterion. Therefore, the *gVent* system was chosen with the knowledge that dynamic aspects would have to be introduced as part of preparing the case study.

The original *gVent* AC argument contains 445 nodes, which is larger than can reasonably be included within this dissertation. A "slice" of the argument focusing on software for controlling the system-level hazard `H-2`: *sustained over-pressure supplied at the patient interface* was chosen. The argument slice, from the top-level claim to the leaves, contains 165 nodes, 112 of which are structural, which satisfies the size criterion for the case selection.

Case Preparation

In addition to selecting a slice of the argument, several steps were taken to prepare the argument for the case study. The argument was developed based on engineering artifacts produced for *gVent*'s proof-of-concept prototype. As a result, the evidence supporting the AC is not as complete as it would be had the project progressed to the point where it was acceptable for use with humans. For transparency and repeatability, the changes made to the *gVent* argument are described below.

Changes to Evidence. To avoid an unnecessarily pessimistic[11] belief valuation, which could block advancement to the Deployment Phase of the *Certus* Method, several additional pieces of evidence were imagined and used to support the argument. Minor adjustments were made to the argument to accommodate this additional evidence (e.g., adding new evidence nodes or adjusting the wording of claims). Care was taken to preserve as much of the original argument as possible during this process. The new evidence was: 1) a system-level design specification, 2) software unit tests, 3) a source code inspection report, 4) a static source code analysis report, and 5) bench test results (i.e., hardware-in-loop tests). Additionally, references to existing pieces of evidence already in the argument were re-used to provide support for claims where relevant. For instance, a software requirements specification (SRS) and system-level test results were included in the original AC and used to provide additional support

for some claims about the correct function of the control software.

Added Indicators. The original *gVent* argument did not include indicators. For the purpose of this case study, it is imagined that *gVent* is deployed as part of a field monitoring program where clinical staff regularly report on the device's usage and provide system logs for inspection by the engineering team. Table 7.9 describes the indicators that are introduced for the case study along with their "Warning" and "Danger" category thresholds.

Table 7.9: Indicators added to the *gVent* case study.

			Thresholds	
Id.	**Name**	**Description**	Warning	Danger
I1	Over Pressure Events (hr^{-1})	The frequency of sustained over-pressure events (`H-2`) reported by clinicians.	$> 10^{-5}$	$> 10^{-4}$
I2	Out of Memory Events (hr^{-1})	The frequency of out-of-memory errors for the *gVent* control software reported in system logs.	$> 10^{-6}$	$> 10^{-5}$
I3	Over-Voltage Events (hr^{-1})	The frequency computing hardware failures due to over-voltage conditions reported in system logs.	$> 10^{-6}$	$> 10^{-5}$

Applying *Certus* in the Case Study

The following describes how the *Certus* Method was applied to the scenario sketched above according to the phases defined in Chapters 2 and 6.

Development Phase. The *gVent* argument existed prior to beginning the case study, so some steps of the method were "compressed" as suggested in Section 6.1.3. Instead of performing Step 1 in parallel with argument development, modelling belief propagation was done in a single pass over the argument. *Certus*' macros were used where appropriate. Since the *gVent* argument used EA, the eliminative variants of the macros was preferred when reasoning steps involved defeaters. Custom expressions, including the patterns described in Section 7.4.2 (belief bounding, conditional propagation, and relevance) were also used. In Step 2, the support of evidence towards parent claims (or rebutting a defeater) was modelled.

[11] Initially I tried assessing belief in the AC "as is" and found that the lack of additional evidence produced inadequate belief valuations.

Evaluation Phase. Since the *gVent* argument already contained defeaters, Step 4 (Model Challenge) was also compressed. As part of this step it was also necessary to briefly return to previous steps to model belief propagation and evidential support for sub-arguments that rebut the defeaters. For Step 5 (Assess Challenge), the credibility of residual doubts were assessed based on my own judgment. Belief valuations for `D135` and `D35` were also conditioned on indicators I2 and I3, respectively. For evidence assessment in Step 6, leaf valuations were assigned based on values in Table 7.10. The evidence was divided into two types: 1) *static* evidence that changes infrequently and is not always (re-)produced for each revision to the system; and 2) *dynamic* evidence that is updated each time the system is changed, potentially produced by automated processes. For dynamic evidence, the belief in the corresponding node in the argument was conditioned on the availability of the underlying artifact. Finally, since an argument slice was used, *high* (`H`) belief was assigned for leaf claims left after their branches were trimmed; sliced branches were marked as "undeveloped".

Table 7.10: Evidence valuations for Step 6 of *Certus* in the *gVent* case study. Valuations and rationale are fictitious.

Name	**Dynamic**	**Valuation**	**Rationale**
System Design Specification	No	*low* (`L`)	The design specification minimally acceptable, but lacks detail in several areas.
Three-Way Valve Specification	No	*certain* (`C`)	Provided by a reliable manufacturer.
Sensor Pressure Specification	No	*certain* (`C`)	Provided by a reliable manufacturer.
Software Requirements Specification	No	*very high* (`VH`)	The specification is of good quality, but records of its verification (reviews etc.) are poor.
Software Architecture Report	No	*very high* (`VH`)	The report is of reasonable quality, but lacks detail in some sections.
Hardware Abstraction Layer Specification	No	*certain* (`C`)	Provided by a reliable manufacturer.
Source Code Inspection Report	Yes	*low* (`L`)	The code inspection was not performed systematically (no checklists etc.).
Software Unit Test Results	Yes	*very high* (`VH`)	The unit tests are of good quality, with appropriate coverage measurements; however, they were not written by an independent test team.
Static Analysis Results	Yes	*very high* (`VH`)	No major problems raised by static analysis, but the static analysis tool is not qualified.
Bench Test Results	Yes	*very high* (`VH`)	All tests have passed, but there is a minor concern about quality of test hardware that might undermine credibility.
System Test Results	Yes	*certain* (`C`)	All tests have passed and all the test procedures and setup are of high quality.

Deployment Phase. In the Deployment Phase, the *high* decision threshold was selected as the operational criteria for Step 8 of the *Certus* Method. Then in Step 9 indicator data is supplied as shown in Table 7.11. Data arrives from field reports spaced 45 days apart. It is imagined that 100 *gVent* units are enrolled in the monitoring program and that each unit is used 50% of the time.

Table 7.11: Indicator values used in the *gVent* case study. Shows six field reports, each with the number of events per report, the number of device hours accumulated since the last deployment, and the rate (hr^{-1}) used to determine threshold category.

		First Deployment			**Second Deployment**		
Indicator	**Value**	R1.1	R1.2	R1.3	R2.1	R2.2	R2.3
Device Hours		*54,000*	*108,000*	*162,000*	*54,000*	*108,000*	*162,000*
I1 (Over Pressure Events)	#	0	0	0	0	0	0
	Rate	0	0	0	0	0	0
	Cat.	-	-	-	-	-	-
I2 (Out of Memory Events)	#	0	1	1	0	0	0
	Rate	0	9.3×10^{-6}	1.2×10^{-5}	0	0	0
	Cat.	-	Warning	Danger	-	-	-
I3 (Over-Voltage Events)	#	0	0	0	0	1	0
	Rate	0	0	0	0	9.3×10^{-6}	6.2×10^{-6}
	Cat.	-	-	-	-	Warning	Warning

7.5.3 Results

The results of the case study are presented over the phases of the *Certus* Method. Samples from the *gVent* argument are used to support discussion. The full argument "slice" for the case study is in Appendix E. The distribution of node types of the original and sliced argument are shown in Table 7.4 along with changes for the modified argument slice that was used for this case study.

Results from the Development Phase

In the Development Phase, belief propagation (Step 1) and evidential support (Step 2) were modelled with a combination of macros, custom expressions, and the default

Table 7.12: Size metrics for the full *gVent* argument and the modified argument "slice" used for the case study.

	Original		**Sliced**		**Modified Slice**			**Rebutted Slice**		
Node Type	#	%	#	%	#	%	Δ	#	%	Δ
Claim	113	25%	39	30%	42	27%	+3	44	27%	+2
Evidence	54	12%	17	13%	33	21%	+16	36	22%	+3
Strategy	26	6%	8	6%	8	5%	0	8	5%	0
Inference	13	3%	3	2%	3	2%	0	4	2%	+1
Complete	46	10%	13	10%	32	21%	+19	35	21%	+3
Assumption	11	2%	0	0%	0	0%	0	0	0%	0
Context	57	13%	7	5%	3	2%	-4	3	2%	0
Defeater	81	18%	20	16%	19	12%	-1	20	12%	+1
Residual	37	8%	9	7%	7	5%	-2	6	4%	-1
Undeveloped	7	2%	13	10%	8	5%	-5	8	5%	0
TOTAL	445	100%	129	100%	155	100%	+26	164	100%	+9
Structural	287	64%	87	67%	105	68%	+18	112	68%	+7
Informative	68	15%	7	5%	3	2%	-4	3	2%	0
Terminal	90	20%	35	27%	47	30%	+12	49	30%	+2

operator. The number of times each type of operator was used in each step is shown in Table 7.13. For instance, to model belief propagation, 30 reasoning steps were considered, of which 6 (17%) required custom expressions, while the remainder were modelled by macros or the default operator (deductive `#FUSE`). The strict macros were not used in this case study and do not appear in Table 7.13.

The top levels of the argument, annotated with propagation expressions and macros, are shown in Figure 7.10. Of note is the expression that conditions the belief in `C13` on the category assigned to indicator I1. The indicator has a value of `0.0` which does not trigger the threshold for either the Warning or Danger categories, so the belief in `S20` is propagated. However, if data were to become available after deployment suggesting the hazard rate exceeds the "Danger" threshold, then belief in the claim would be bounded to, at most, *skeptical* by: `min(skep,S20)`.

When modelling evidential support, the most common operator was the default deductive `#FUSE` macro, which computes an average among child beliefs. Belief bounding, as described in Section 7.4.2, was occasionally used, but mainly for argument

Table 7.13: Summary of *Certus* propagation operators applied to the *gVent* case study. Includes macros, custom ("Cust."), and default ("Def.") propagation operators, and direct and expression-based leaf assignments.

		Deductive			Eliminative					Leaf	
Certus Step	Steps	FUSE	MIN	MAX	FUSE	MIN	MAX	Cust.	Def.	Direct	Expr.
Model Belief Prop.	30	0	6	1	0	1	0	6	16	0	0
Model Support	19	0	0	0	1	1	0	3	14	0	0
Model Challenge	11	0	0	0	2	6	0	3	0	0	0
Assess Challenge	0	0	0	0	0	0	0	0	0	5	2
Assess Evidence	1	0	0	0	0	0	0	1	0	9	22
Rebuttal	5	0	0	0	1	0	0	1	3	2	2
TOTAL	66	0	6	1	4	8	0	14	33	16	26

steps containing both defeater- and evidence-typed children.

Results from the Evaluation Phase

The Evaluation phase contained several steps that culminated in the decision to deploy the *gVent* system. To begin, the deployment threshold of *high* (`H`) was chosen for the argument. The subsequent paragraphs describe the results of applying the remaining steps in the Evaluation phase.

Step 4: Model Challenges. The impact of the argument's 19 defeaters was modelled across 11 argument steps. An example of this appears in Figure 7.11, where two defeaters, `D125` and `D133`, challenge the parent node, `C377`, and the `#MIN_ELIM` macro is used to determine the belief in the parent. The `#MIN_ELIM` macro selects the minimum among the inverted child beliefs. Another example is the impact of `D135` on `IR140`, which uses inverted belief bounding to limit the maximum belief permitted for `IR140`, even if `D135` is entirely rejected; this is intended to account for the possibility that the challenge against `IR140` is incomplete. Table 7.13 summarizes the use of macros, custom expression, and default propagation operators in the model challenge step and then in the subsequent rebuttal.

Step 5: Assess Challenge. Each residual doubt was either assigned a belief valuation indicating the credibility of the doubt, or its belief was conditioned on an indicator. Figure 7.11 shows the later case, where indicator I2 (Out of Memory

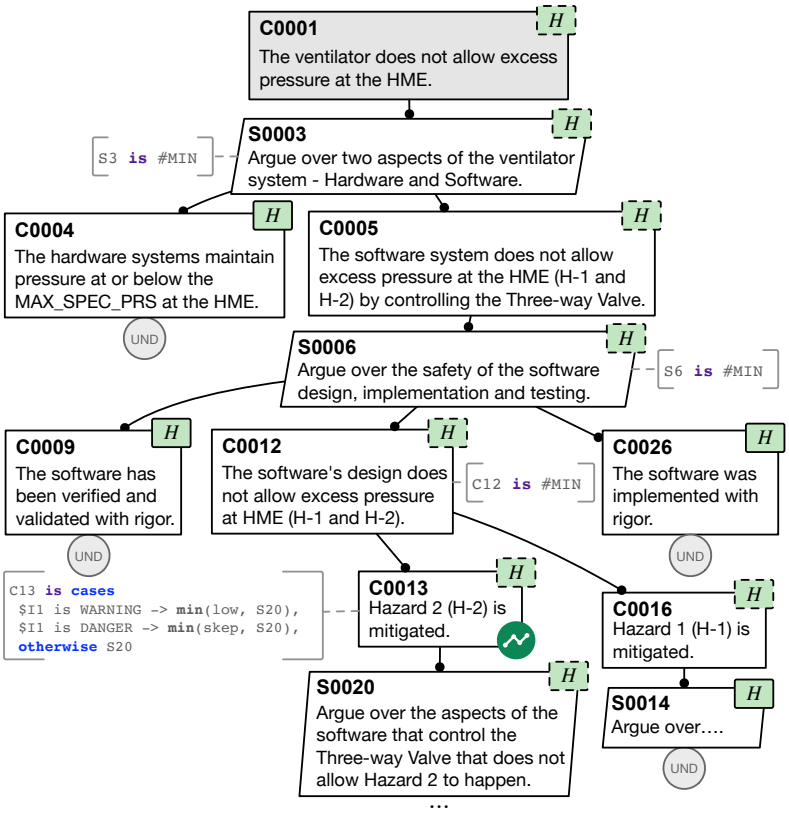


Figure 7.10: Sample of results from applying Step 1 of the *Certus* Method to the *gVent* case study. Annotated with non-default propagation operators, indicator I1 on `C13`. Sliced branches are shown with undeveloped nodes (`UND`). Continues below `S20`

Events) is used to determine the credibility of `D135`. Before system deployment no data is available, so the indicator's value is `0.0` and the credibility of `D135` is taken as *very opposed* (`VO`).

Step 6: Assess Evidence. The evidence valuations from Table 7.10 were used to assign belief levels for leaf nodes. For non-dynamic evidence, values were directly assigned, like for `E141` in Figure 7.11. For dynamic evidence, the belief was conditioned on the availability of the related artifact. For `E138`, the belief is taken as *low* (`L`) when the Inspection Report artifact is available, but the evidence is rejected if the inspection report is not available.

While selecting belief valuations for the evidence, an interesting trade-off was repeatedly observed. The reasoning from Table 7.10 justifies belief valuations that are less than *certain*. The rationale describes deficiencies with the evidence or its production. Instead of encoding these deficiencies as lower belief values, they could

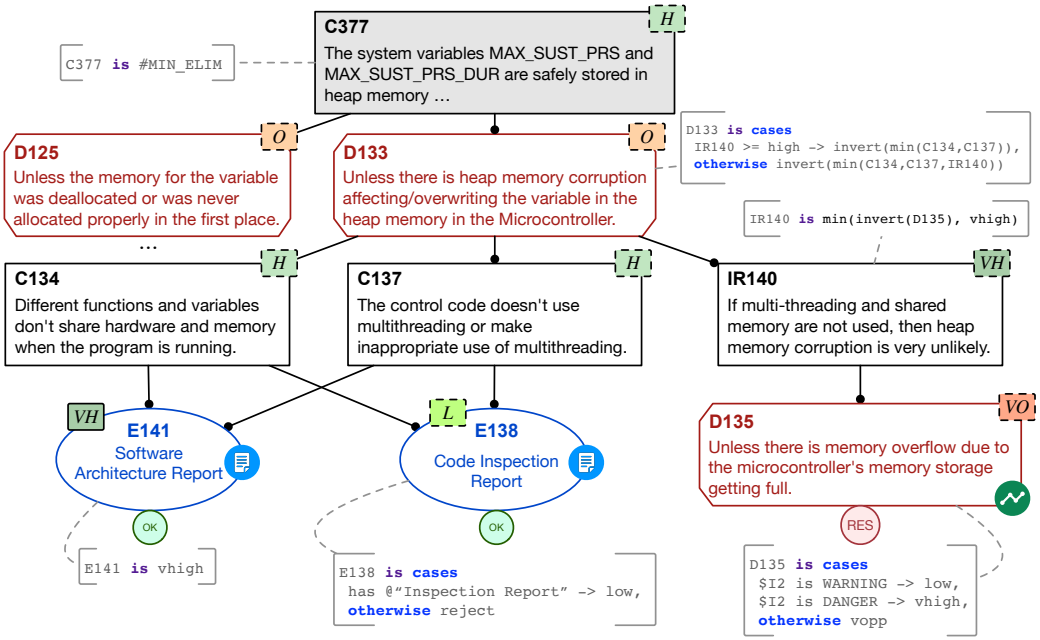


Figure 7.11: Sample of results from the Evaluation phase of the *Certus* Method on the *gVent* case study. Before reports of out of memory errors for indicator I2.

have been represented as undermining defeaters that challenge the evidence nodes in the AC. This is a manifestation of an open problem with the dialectic approach: there are no clear stopping rules for identifying defeaters. This case study revealed a (partial) answer to this problem by providing an alternative means to account for any remaining uncertainty in the evidence.

Step 7: Decide on Deployment. The overall belief computed by *Certus* for the (slice of) *gVent* argument was *high* (H), which satisfies the deployment criterion from Step 3. That is, for the purpose of the case, there was sufficient belief in the AC to deploy the *gVent*.

Results from the Deployment Phase

The Deployment phase begins with determining criteria to continue operating the system. For *gVent*, the operational criteria are the same as the deployment criteria: belief in the AC's top-level claim must be at least *high*. Following the case study narrative from Section 7.5.1, after 90 days of operation, field reports arrived suggesting that Out of Memory errors were occurring with a frequency exceeding the "Warning" threshold for indicator I2. The belief level in the top-level claim drops to *skeptical* (SK), which is below the operational criteria. A notice is sent to clinicians using

gVent advising them to suspend its use, or restart the device every 24 hours, until the problem can be resolved.

Results for System Evolution and Re-Deployment

After *gVent*'s deployment was suspended, the software was patched to address out of memory events. In the updated argument, rather than being residual, `D135` has a rebutting sub-argument that relies on unit test results and updates to the software architecture and specification documents. The indicator's value is reset to `0.0` since the memory problem has been fixed. The *Certus* Method is repeated on a small scale for this sub-argument, with size and operator metrics shown in Tables 7.12 and 7.13.

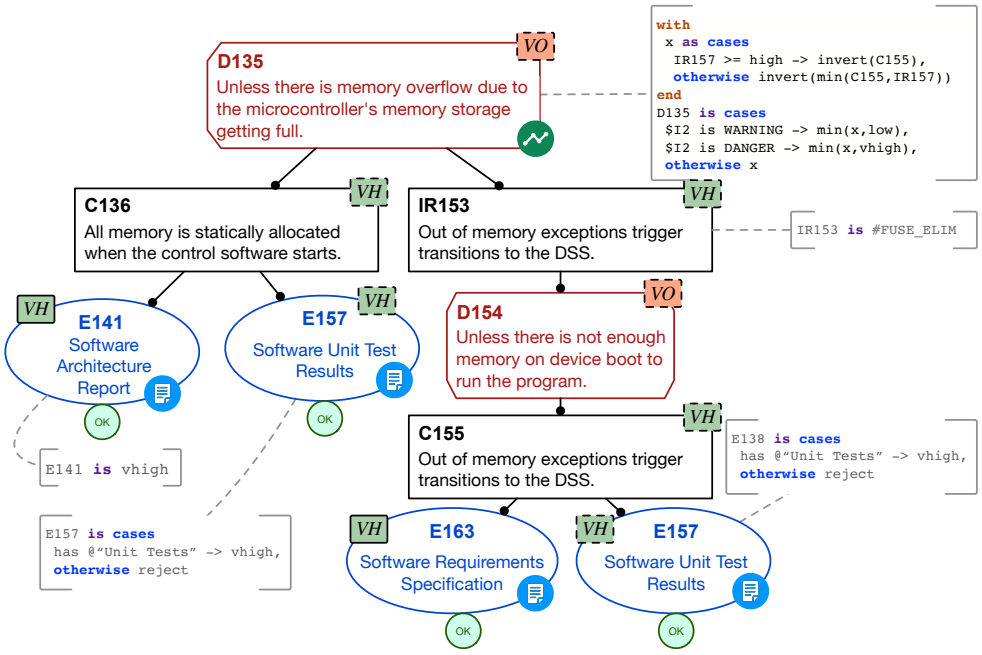


Figure 7.12: Rebuttal to `D135` added in response to out of memory errors.

In the case study narrative, the *gVent* team considers releasing the system without repeating a number of verification activities. The absence of this evidence results in the overall belief being insufficient to re-deploy. For example, if the System Test Report is not supplied to *Certus*, the top-level claim is evaluated as *reject*, and if the source code inspection report is not provided, then the result is *uncertain*. If all evidence is updated and provided, then top-level claim is evaluated to have *high* belief and the revised system may be re-deployed.

After re-deployment, the case study's narrative suggests that some over-voltage

conditions are reported. These trigger indicator I3 to exceed the "Warning" threshold and cause *Certus* to output a *skeptical* valuation for the top-level claim. The decision is made to suspend clinical operations until the hardware problem can be fixed.

7.5.4 Discussion

This case study has provided an end-to-end example of using *Certus* to assess belief in a DAC for a critical system. Two iterations of the *Certus* method were applied, one for pre-deployment development, and again in response to field reports of software failures. The results demonstrate that *Certus* is *applicable* to real-world DACs, including during the Deployment phase of the AC method and for system evolution within a broader process of continuous assurance (RQ1).

The case study also explored situations where the overall belief in the argument changed based on indicator values and the presence of supporting artifacts. That is, the DAC was shown to be sensitive to these dynamic aspects (RQ2). After the first deployment of *gVent*, when Indicator I2's warning threshold was exceeded, *Certus*' belief valuation was updated to show that belief in the top-level claim was not sufficient to continue operations. Then, even after the problem was addressed, missing artifacts resulted in an insufficient belief in the top-level claim. In both cases, the AC was shown to be sensitive to changes in the dynamic aspects of the argument.

The analysis in Section 7.4 showed that *Certus* can express a reasonable range of propagation operations that might be encountered in AC arguments. The results from this case study provide further evidence of *Certus*' expressivity on an AC argument for a real-world system (RQ3). Moreover, the case study showed that indicators and artifacts can be incorporated into *Certus* expressions.

Finally, inspecting Table 7.13 provides insight into the level of effort required to apply *Certus* to the case study (RQ4). Using the data from Table 7.13, the decision complexity model from the scalability analysis in Section 7.2 yields a total decision complexity of $\Omega_d = 144$ decisions. Notably, this is close to the predicted number of 123 decisions from Table 7.5. While applying *Certus*, the number of decisions did not feel onerous, taking on the order of 4 hours to complete the case study's belief assessment spread across two working sessions. Overall, applying *Certus* did not appear to be a particularly time or effort intensive task.

Limitation: Case Study Realism. To fully demonstrate the *Certus* Method on a DAC, this case study mixed real and fictitious elements. Since development of

the *gVent* system stopped after a proof-of-concept prototype was created, it was necessary to imagine additional pieces of evidence that would have been produced if system development had continued. The impact of this was limited by selecting typical types of evidence that would be encountered while developing safety-critical software. Second, the dynamic elements of the AC (indicators and artifacts) were imagined, including indicator data from field reports. On one hand, these imagined elements undermine the claims made in the discussion above regarding realism. On the other hand, the use of an AC for *gVent*, a real system helps to ground the imagined aspects of the case study in a real-world context.

Limitation: Confirmation Bias. The case study, including the belief assessment, was performed by one person (myself) which means that confirmation bias cannot be ruled out as a source of error. Applying the *Certus* Method systematically, and reporting detailed results above (and in Appendix E) helps to mitigate this limitation.

7.5.5 Conclusion

This case study has provided an end-to-end demonstration of the *Certus* language and method on a DAC that was developed for (a prototype of) a critical medical device. Accounting for the limitations, the results of the case study provide additional evidence that: *Certus* is applicable to DACs for critical systems; its belief assessments are sensitive to dynamic aspects of the DAC; the language is expressive enough to be applied to real-world arguments; and the level of effort required to apply the method is reasonable.

7.6 Satisfaction of Requirements

Chapter 4 defined 24 requirements for a new CAM from the practitioner, dynamic, and dialectic perspectives. This section considers the *Certus* language, method, and their evaluation presented thus far to determine if the requirements are satisfied.

7.6.1 Method

Each requirement in Table 7.14 was reviewed and assigned a satisfaction level:

- **Not Satisfied** ("−") - The *Certus* language and method do not satisfy the requirement.
- **Partially Satisfied** (◯) - The *Certus* language or method partially satisfy the requirement, but limitations exist that could be addressed by future work.
- **Satisfied in Principle** (⊙) - The definition of the *Certus* language or method satisfy the requirement, but additional evidence is not available to confirm the requirement is satisfied.
- **Fully Satisfied** (⊕) - The definition of the *Certus* language or method satisfies the requirement in principle and there is additional evidence confirming the requirement is satisfied in the form of an experiment, tests, or case study.

Several of the requirements from the dialectic perspective (R2-1 to R2-11) can only be evaluated for *Certus*' macros. For instance, R2-1 considers the situation where a credible defeater is added under a claim. A custom *Certus* expression could be crafted to ignore the defeater.

7.6.2 Results

Table 7.14 shows the results of the satisfaction analysis for each requirement. References are provided to relevant sections to substantiate the rationale.

Table 7.14: Satisfaction of the requirements for a new CAM from Chapter 4 by *Certus*. The symbol ⊕ means the requirement is satisfied with supporting evidence, ⊙ means the requirement is satisfied in principle, ◯ means the requirement is partially satisfied, and "−" means requirement is not satisfied.

Perspective	Id.	Req. Name	Sat.	Rationale
Practitioner	R1	Structured Argumentation	⊕	*Certus* is intended for use with structured argument notations. Confirmed by inspection of the language and method definition in Sections 5.4 and 6.1, by the expressivity analysis in Section 7.4 and the case study in Section 7.5.
	R2	Defeaters	⊕	Both the *Certus* language and method has support for dialectic reasoning. Defeaters can be included in custom expressions in the language and are accounted for in built-in macros. Steps 4 and 5 of the method account for defeaters. Confirmed by the expressivity analysis in Section 7.4 and the case study in Section 7.5.
	R3	Subjectivity	⊙	*Certus* mitigates subjectivity by using fuzzy sets to model belief, enabling vague, but linguistically meaningful, expressions of belief. *Certus* then uses fuzzy sets to define several belief levels (i.e., the canonical sets). Satisfied in principle by the definition of the fuzzy sets in Section 5.3, but has not been empirically validated.
	R4	Interpretation	⊙	The names of *Certus*' canonical sets align with belief levels that are readily interpreted by a wide range of interest holders (e.g., *skeptical* vs. *very high* belief). Satisfied in principle by the definition of the fuzzy sets in Section 5.3, but has not been empirically validated.

Perspective	Id.	Req. Name	Sat.	Rationale
	R5	Understandability	$\odot$	The *Certus* language defines belief propagation using expressions that can be interpreted in natural language so that users can understand how belief is propagated through the argument. This requirement is satisfied in principle by the definition of the language's syntax. Further, the propagation analysis in Section 7.1 provides some empirical support by confirming that belief propagation in *Certus* is not susceptible to attenuation or amplification effects, which makes results harder to understand. However, understandability has not been empirically validated with end-users.
	R6	Expressivity	$\oplus$	The *Certus* language is designed to offer flexibility in the expression of belief propagation operations, including capturing nuanced propagation logic (e.g., conditioning). The expressivity analysis in Section 7.4 and case study in Section 7.5 confirm that the language can express a reasonably wide range of propagation expressions from numerous ACs.
	R7	Applicability Criteria	$\oplus$	Each step in the *Certus* Method defines pre-conditions that must be satisfied prior to its application. Confirmed by inspection of the method in Chapter 6 and by application of the method to a case study in Section 7.5.
	R8	Stopping Criteria	$\oplus$	Each step in the *Certus* Method defines post-conditions that should be satisfied when the step is complete. Confirmed by inspection of the method in Chapter 6 and by application of the method to a case study in Section 7.5.
	R9	Effort	$\oplus$	In the average case, *Certus* requires users to make fewer decisions to apply than other methods. Confirmed by the scalability analysis in Section 7.2, and supported by additional evidence from the expressivity analysis in Section 7.4 and the case study in Section 7.5.

Perspective	Id.	Req. Name	Sat.	Rationale
Dynamic	R10	Evolving Arguments	◯	When the argument changes, *Certus* can be used to automatically re-evaluate belief in the argument. However, custom *Certus* expression might need to be manually updated to account for the change. The formal syntax semantics of *Certus* provide a basis for automated updates expressions, but automated procedures for adapting belief expressions have not been developed. *Certus*' macros are less sensitive to changes in the argument structure and are preferred to mitigate the impact of change. This requirement is partially satisfied by the definition of the language in Chapter 5 and the method in Chapter 6. The case study in Section 7.5 also included a modest change to the argument.
	R11	Updated Evidence	⊕	*Certus* can condition belief valuations on the availability of supporting artifacts. When an AC is changed as part of a dynamic assurance process, if an artifact is not available, then the missing evidence is accounted for in the belief computation. If evidence is re-assessed, then *Certus* can re-compute the belief in the argument to determine if it is acceptable. Satisfied in principle by *Certus* language specification in Section 5.7 and demonstrated by the case study in Section 7.5.
	R12	Indicators	⊕	*Certus* can condition belief valuations on indicators. When their values change, the belief in the argument can be re-evaluated and used to determine if a system should continue to operate. Satisfied in principle by the language specification in Section 5.6, the method in Chapter 6 and demonstrated by the case study in Section 7.5.

Perspective	Id.	Req. Name	Sat.	Rationale
	R13	Decidability	⊕	If leaf assessments are provided by users, *Certus* can automatically re-evaluate belief in an argument, which can be used as a basis for decisions to deploy, or continuing to operate, a system. Satisfied in principle by belief propagation procedure in Section 5.4.7, the method steps in Section 6.3, and demonstrated by the case study in Section 7.5.
Dialectic	R2-1	Adding Credible Defeaters	⊕	*Certus*' macros are sensitive newly added defeaters up to the underlying intent of the macro's operation. Satisfied in principle the definition of the macros in Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.
	R2-2	Decomposing Defeaters	⊙	*Certus*' macros support cases where defeaters are decomposed into sub-defeaters. Satisfied in principle by the definition of the macros in Section 5.5. However, an argument fragment that had a defeater decomposed by another defeater was not part of the expressivity analysis or case study. Both unit and manual testing of the `certus-ts` implementation show the implementation satisfies this requirement.
	R2-3	Weighting Defeaters	⊕	Custom expressions can be composed in *Certus* that describe the relative importance between child nodes, including among child defeaters. Satisfied in principle by the language's definition in Chapter 5 and confirmed in the expressivity analysis in Section 7.4.
	R2-4	Resolving Defeaters Increases Belief	⊕	*Certus*' macros are sensitive to resolved defeaters, up to the underlying intent of the macro's operation. Macros are available for three different interpretations of dialectic reasoning. Satisfied in principle by the definition of the macros in Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.

Perspective	Id.	Req. Name	Sat.	Rationale
	R2-5	Resolving All Defeaters	⊕	As above. Satisfied in principle by Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.
	R2-6	Denoting Resolved Defeaters	⊕	Satisfied by the implementation of *Certus* the *Socrates* tool as described in Section 5.8. Denoting resolved defeaters is related to the visualization of assessment results, and is best satisfied as part of the user interface for tools supporting the method. This requirement was confirmed in the case study in Section 7.5.
	R2-7	Residual Defeaters	⊕	Residual defeaters are leaf nodes in the argument. Per the *Certus* Method they should be assigned a belief valuation by the user. Satisfied in principle by Step 5 and confirmed by the case study in Section 7.5.
	R2-8	Increasing Defeater Belief	⊕	*Certus*' macros are sensitive to increases (and decreases) in the belief of credible child defeaters and claims, up to their underlying definition. Satisfied in principle by the macro definitions in Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.
	R2-9	Decreasing Defeater Belief	⊕	As above. Satisfied by the macro definitions in Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.
	R2-10	Increasing Claim Belief	⊕	As above. Satisfied by the macro definitions in Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.
	R2-11	Decreasing Claim Belief	⊕	As above. Satisfied by the macro definitions in Section 5.5 and confirmed by the sensitivity analysis in Section 7.3.

7.6.3 Discussion

All 24 requirements identified in Chapter 4 were at least partially satisfied: overall, this suggests a positive outlook for *Certus*. Of the requirements, 19 were fully satisfied, 4 were satisfied in principle, and one was partially satisfied. The requirements that were not fully satisfied represent significant challenges for CAMs more broadly. That is, it is not surprising that *Certus* does not immediately satisfy them and that more work in these areas is required. These are discussed further below.

Additional Empirical Evidence. Three requirements from the practitioner perspective, R3 (Subjectivity), R4 (Interpretation), and R5 (Understandability), were satisfied in principle by *Certus*: the language was designed with these properties in mind. For instance, fuzzy sets expressed over a well-defined belief scale were used to improve the interpretability of the belief valuations. Preliminary results from the expressivity analysis and case study are promising. However, since these requirements are fundamentally about the interaction between *Certus* and human users, empirical evidence generated by multiple users applying *Certus* is required. User studies or a collection of case studies might provide suitable evidence.

Handling Evolving Arguments. R10 (Evolving Arguments) was the only partially satisfied requirement. When an argument's structure changes, *Certus* has some ability to tolerate the change (e.g., macros) and can re-compute the belief levels once any custom expressions are updated. However, a user must still update custom expressions and validate that choices for macros are valid. Change management for AC arguments remains an active area of research. While *Certus*' formal syntax and semantics provide the foundations of a confidence change management approach, automated procedures for revised belief propagation expressions have not been developed.

7.7 Chapter Summary

This chapter has presented a series of evaluation activities that collectively demonstrate that the *Certus* language and method have desirable properties and that satisfy, in full or in part, the requirements for a new CAM defined in Chapter 4. More specifically, the evaluation activities from this chapter show that:

- Belief propagation using macros in *Certus* is not susceptible to belief attenuation and amplification effects that impact other quantitative CAMs.
- In the average case, belief assessments using *Certus* require fewer decisions on the part of the user (i.e., less effort) than for other quantitative CAMs.
- *Certus*'s macros are appropriately sensitive to changes in belief among child nodes, including for argument steps that involve defeaters.
- *Certus* can express a wide range of belief propagation operations that commonly appear in AC arguments.
- *Certus* is applicable to ACs that are developed for real-world systems.
- *Certus* is applicable to DACs that are embedded as part of a continuous assurance process. These DACs might include dynamically linked artifacts or indicators derived from operational data.

The next chapter will discuss these evaluation results as they relate to the overall research goal of this dissertation.

Chapter 8
Conclusions and Future Directions

Preparing an assurance case (AC) is an important activity for determining if a critical system can be deployed or continue to operate. System developers increasingly depend on Artificial Intelligence, self-adaptive architectures, and continuous assurance processes to manage uncertainty in operational environments. The goal-oriented approach offered by ACs has become an important part of the strategy for assuring these types of systems. Moreover, it is no longer sufficient to create a "static" AC as part of wrapping up system development. To provide *through-life* assurance, ACs must become dynamic models for making assurance-related decisions. These *dynamic assurance cases* (DACs) can be instrumented with "live" data streams, linked to dynamic evidence produced by continuous development and integration (CI/CD) pipelines, and modified during system evolution. Each change to the argument (or its support) should trigger a re-assessment of the DAC to determine whether its claims are satisfied. Several methods for assessing confidence in ACs have been proposed, and some have been adopted by practitioners. However, challenges remain that limit the adoption of these methods and make them difficult to apply to DACs, especially for quantitative assessment methods.

8.1 Research Goal Revisited

The overall goal of this research has been to create a new confidence assessment method (CAM) that: is applicable to DACs, addresses barriers to adoption facing existing CAMs, has support for dialectic reasoning, and produces trustworthy results that can be relied upon by practitioners. Towards this goal, the research for this dissertation began with a series of interviews that were used to develop the "Prac-

titioner Perspective" on confidence assessment, including identifying limitations of existing CAMs. Building on the results of the interviews, a set of requirements was specified that were used as the basis for a new CAM.

Next, this dissertation developed the *Certus* language and method for assessing belief in DACs. *Certus* is a mixed (qualitative and quantitative) CAM that uses fuzzy sets to express and manipulate linguistically meaningful valuations of belief. With the language, expressions are created to describe how belief propagates through an argument, offering a high degree of flexibility to users. The *Certus* method describes how to apply the language to assess belief in a (D)AC, including integrating "live" data from indicators and dynamic sources of evidence produced by continuous or perpetual assurance processes. The following describes how *Certus* addresses the detailed criteria for the research goal set out in Section 1.6.

8.1.1 Addressing Limitations of Quantitative Methods

Since *Certus* is a mixed CAM, it is vulnerable to issues that have limited the adoption of quantitative methods, including: subjectivity of assessments, interpretation of results, over-precision fallacies, inflexibility when expressing nuanced reasoning, the level of effort required to apply the method, and selecting decision thresholds. *Certus*' approach to addressing these challenges is discussed below.

Subjectivity, Interpretation, and Over-Precision

In principle, *Certus*' linguistic representation of belief (underpinned by fuzzy sets) addresses challenges related to subjectivity, interpretation of results, and over-precision. Users express their belief using phrases like "*very high*" or "*opposed*", reducing the need to select specific numbers with arbitrary precision. Describing belief linguistically also helps to communicate assessment results to interest holders. Various examples, including a larger-scale case study, within this dissertation have shown that it is possible to represent belief in this way. However, the hypothesis that linguistic reasoning helps to mitigate these challenges has not been extensively validated (e.g., through user studies), which is an important limitation of this work.

Flexibility in Reasoning

Flexibility and expressivity were key considerations when designing the *Certus* language. Indeed, users can describe how belief propagates through an argument in

at least three ways: user-defined operators, built-in macros, or custom propagation expressions. The expressivity of the language was validated in a dedicated analysis and as part of a case study. While it is possible that arguments exist that cannot be reasonably modelled in *Certus*, the results from Chapter 7 suggest that *Certus* is capable of modelling belief propagation for reasoning steps that commonly arise in AC arguments.

Support for Decision-Making

Since ACs are used to make decisions about whether to deploy a critical system, it is important that CAMs offer support for decision-making. For some DACs, it is desirable to have (semi-)automated decision processes that can be re-run after the DAC changes. A benefit of *Certus* is that the language's evaluation procedure can be used to automatically (re-)compute belief assessments, including after evidence is updated or an indicator's value has changed. This enables run-time decision-making about system assurance.

A key aspect of decision-making is selecting the minimum belief level that must be achieved (or maintained) to operate a system, i.e., a "decision threshold". The *Certus* method includes steps that require a user to define, and then use, these thresholds for deployment decisions. In principle, the ability to express belief linguistically also helps when selecting decision thresholds, since thresholds can be described as fuzzy sets rather than as a single number. While the decision-support features of *Certus* were demonstrated as part of a case study, the absence of empirical data to show that linguistic decision thresholds are interpretable and useful for practitioners (and their interest holders) is a limitation of this research.

Level of Effort

The level of effort required to apply a CAM is a common concern among practitioners. Indeed, some additional effort is required, compared to not performing a confidence assessment at all. In the worst case, where complex belief propagation operations are required at every step in an argument, applying *Certus* requires a significant amount of effort. However, the *Certus* language includes several features that reduce effort, including pre-defined macros, re-usable belief propagation operators, and sensible defaults. Results from the scalability analysis and case study in Chapter 7 show that these features reduce the level of effort required to apply *Certus* to a manageable

level in the average case. Of course, there remains an open question as to whether the outcome of applying *Certus* is worth the effort. Fully answering this question would require long-term case studies showing that *Certus* provides value as part of the development and maintenance of real-world (D)ACs.

8.1.2 Support for Dialectic Reasoning

Given the growing interest in dialectic reasoning, it was important that *Certus* support belief assessment of (D)ACs that contain defeaters. This was achieved in several ways. First, the *Certus* method contains explicit steps to model the impact of defeaters on belief propagation and to provide belief valuations for unresolved defeaters. Second, the language's built-in macros can be applied to reasoning steps that contain defeaters. The macros encode three different interpretations of dialectic reasoning that a user may select from. Finally, *Certus* can be used to condition the belief in defeaters on the value of indicators derived from "live" data collected from system development activities or operations. The sensitivity analysis, expressivity analysis, and case study from Chapter 7 provide evidence of *Certus*' support for defeaters.

8.1.3 Support for Dynamic Assurance

While *Certus* can be used to assess belief in static ACs, some of its capabilities were developed specifically for dynamic ACs. The *Certus* language can be used to condition belief in the argument on both the value of indicators and the presence (or absence) of supporting artifacts. Additionally, the method's description explicitly considers whether indicators or artifacts should be included in belief valuation. As mentioned above, *Certus* also supports (semi-)automated decision-making, which is important for continuous or perpetual assurance processes, e.g., as part of a self-adaptive system's adaptation procedure.

One aspect of dynamic assurance that *Certus* does not fully address is the automated management of change in the structure of a DAC. For instance, depending on the belief propagation operation, children added under an argument step might not be automatically accounted for in the belief assessment. If a macro is used, then its expansion will take account of the new node; however, custom expressions provided by users might need to be updated. A change scenario was considered in the case study in Chapter 7, though this was illustrative rather than describing a complete change management solution. For the time being, this limitation is tolerable because

substantive changes to a DAC's argument structure should not be taken lightly: when the assurance rationale for a critical system changes, a human expert should be involved in determining whether the change is acceptable. If automated procedures are to be used (e.g., an adaptation procedure for a self-adaptive system), then the adaptation would need to consider the impact of changes on confidence (*Certus* or otherwise).

8.1.4 Trustworthiness

Since CAMs are used to make decisions about critical systems, it is important that they produce trustworthy results [80]. Chapter 1 identified five elements that contribute to a CAM's trustworthiness: applicability, understandability, sensitivity, propagation stability, and intuitiveness. The applicability of *Certus* to (D)ACs is discussed from several perspectives above, and is supported by both the case study and the expressivity analysis in Chapter 7. The remaining elements of trustworthiness are considered below.

Understandability

For a CAM to be trustworthy, the mechanics of how it propagates belief through an argument must be readily understood by practitioners. There are two perspectives on understandability to consider.

From a language-theoretic perspective, the syntax and semantics of *Certus* are unambiguously defined. Much of Chapter 5 was dedicated to formalizing the language using the Lean4 theorem prover, and this specification is provided in full in Appendix B. Because the language has a formal semantics, sentences generated by the language's grammar can be reduced to a fuzzy set over a joint possibility-necessity scale, thus giving each sentence an unambiguous meaning. Any formally inclined user (or researcher) can determine the precise meaning of expressions written in the language.

From an end-user perspective, an important feature of the *Certus* language is its ability to describe propagation logic with expressions that resemble natural language. In principle, this increases the understandability of belief propagation operations by allowing, even novice users, to inspect the propagation logic. Even *Certus*' macros, whose treatment of defeaters is arguably the most conceptually complex part of the language, can be expanded into simple, albeit large, `cases` expressions that can be

inspected to understand their behaviour. Nonetheless, the understandability has not been empirically validated (e.g., with a user study), which is a limitation of this research.

Sensitivity

For large ACs, it is possible that the impact of an essential piece of evidence or residual defeater in the leaves of the argument is not "felt" at the top-level claim of the argument. That is, the belief assessment might not be sensitive to important aspects of the argument. With *Certus*, it is possible to create propagation operations that are sensitive to belief valuations at the leaves of the argument (e.g., using the `#MIN` macro). The sensitivity of *Certus*' macros was evaluated in Chapter 7 with a dedicated sensitivity analysis and then further demonstrated as part of the case study. Of course, it is possible to create belief propagation expressions, and transitively whole arguments, that are not sensitive (or only partially sensitive) to certain nodes, but this would be an explicit modelling choice made by the user.

Propagation Stability

Some quantitative CAMs are vulnerable to belief attenuation and amplification effects when applied to arguments that are very deep or broad. This can result in unintuitive outcomes where the belief computed for the top-level claim in the argument is not consistent with the belief valuations provided at the leaves of the argument by the user. Chapter 7 included a dedicated propagation analysis showing that *Certus*' macros are not vulnerable to the belief attenuation and amplification effects that affect other quantitative CAMs.

Intuitiveness

An important aspect of trustworthiness is whether a CAM produces belief valuations aligning with expert intuition and judgement. The case study in Chapter 7 provides preliminary evidence that *Certus* produces belief assessments that align with intuition. However, aside from mitigating the impact of propagation effects (noted above), this dissertation has not systematically evaluated the intuitiveness of belief assessments produced by *Certus*. Indeed, this property might be difficult to measure, even in a controlled experiment.

8.2 Summary of Key Results

The following are the key results from the research described in this dissertation:

- The perspective of AC practitioners on confidence assessment was articulated. Several important barriers limiting the adoption of CAMs for real-world (D)ACs were identified.
- Dialectic reasoning (i.e., "defeaters") can be incorporated into quantitative confidence assessment methods.
- There are at least three interpretations of dialectic reasoning that modify how defeaters are viewed: the deductive, eliminative, or strict interpretation.
- Confidence in a (D)AC may be described using vague, but linguistically meaningful, expressions that can be modelled with fuzzy sets to connect qualitative and quantitative valuations of confidence (resp. belief).
- Confidence propagation in an AC's argument can be described using a domain specific language, which enables the expression of nuanced reasoning.
- Confidence assessments can be conditioned on dynamic elements (e.g., indicators, artifacts), thus enabling (re-)assessment as part of continuous or perpetual assurance activities.

8.3 Future Directions

There are at least four areas of future work that can build on the contributions of this dissertation: additional validation of the core ideas underpinning *Certus*; additional validation of the *Certus* language and method; potential extensions of *Certus*; and knowledge transfer to practitioners.

8.3.1 Additional Conceptual Validation

Among the conceptual contributions listed above, there are at least four lines of inquiry to explore in future work. First, the practitioner perspective on AC confidence assessment should be developed further, using additional studies (e.g., questionnaires) to cross-validate the results of the practitioner interviews presented in Chapter 3.

Second, the notion that belief in an AC can be represented linguistically should be studied further. In particular, the hypotheses that linguistically representing belief aids with subjectivity and interpretation issues, and mitigates over-precision fallacies (compared to purely quantitative representations) should be examined. Since these properties are inherently related to the use of belief assessments by humans, user studies or a series of additional case studies (involving practitioners) would be required to validate these hypotheses. Third, additional studies demonstrating that belief assessments can be performed for DACs should be undertaken, and further research is required to develop methods that update belief assessments when the structure of an AC is changed as part of dynamic assurance activities. Finally, it might be beneficial to extract principles for quantitative confidence assessment that can be applied across methods.

8.3.2 Additional Validation of *Certus*

This dissertation has introduced the *Certus* method and language. While the results of evaluating *Certus* are promising, additional validation is required.

First, the expressivity of the *Certus* language should be measured using a wider range of argument samples. Expressivity should also be compared with other CAMs (e.g., the BBN- or DST-based methods) to determine whether *Certus* is capable of expressing belief propagation operations that other CAMs cannot (or vice-versa). As part of investigating expressivity, commonly occurring belief propagation operators could be identified, which would be candidates for new *Certus* macros.

Second, additional validation activities should be undertaken to establish the trustworthiness of *Certus*. Two properties are of particular interest: understandability and intuitiveness. Additional studies are required to show that *Certus*' propagation operations are understandable to end-users, possibly comparing the understandability between *Certus* and other CAMs. The intuitiveness of *Certus*' belief assessments must also be examined. Part of this work might require defining what it means for a CAM to produce intuitive results, which would be a precondition for measuring this property as part of user studies. Other properties related to trustworthiness might also be identified and evaluated for *Certus*.

Finally, it could be interesting to study how *Certus* can interface with other qualitative or quantitative CAMs. For quantitative CAMs, it might be possible to make numerical transformations to move from the domain of fuzzy sets to that of prob-

ability theory or Dempster-Shafer Theory. This would enable users to assess belief using different methods for different parts of an AC. For example, one sub-argument might be amenable to probabilistic reasoning offered by the BBN method whereas others might benefit from the linguistic valuation of belief offered by *Certus*. For qualitative methods (e.g., ACPs [47] or iTest [125]), it could be valuable to develop a methodology that shows how *Certus* can interface with these methods. The assessment criteria of the iTest method could be used as a guide for crafting propagation logic or providing leaf valuations in *Certus*.

8.3.3 Potential Extensions

From a more technical perspective, there are numerous extensions to the *Certus* language that might enable more sophisticated belief propagation expressions or improve usability. Some potential avenues include:

- Developing a declarative notation for visually describing propagation operators (macros or custom logic). This would lead to more visually concise diagrams compared to the diagrams that appear in this dissertation.
- Providing a more sophisticated artifact assessment mechanism that looks beyond whether the artifact is available (or not). The concepts related to "checkable safety cases" developed by Carlan's dissertation might be helpful [219].
- Providing guidance (in the *Certus* method) and technical interfaces (in the language) for AC change management for common types of changes to AC arguments (e.g., adding a child node, deleting a child node, etc.).
- Developing static analysis capabilities that check whether custom expressions written in *Certus* have specific properties, such as whether they handle defeaters according to the requirements for incorporating defeaters into quantitative CAMs defined in Chapter 4.
- Developing analysis capabilities to perform sensitivity analyses on arguments that contain belief propagation expressions to determine to what extent an argument is sensitive to a specific piece of evidence. An analysis procedure could also be developed to determine the maximum or minimum belief level that can be produced by the combination of a specific argument and belief propagation operators.

8.3.4 Knowledge Transfer to Practitioners

In [52], we observed that the successful adoption of CAMs into engineering practice requires that CAMs be "crystallized" into actionable methods, and that guidance be "curated" describing how to apply CAMs in practical settings. As a start, an open-source implementation of *Certus* has been created and has been incorporated, as a prototype feature, into a commercial AC tool, *Socrates*. However, more work remains in terms of making the method accessible to practitioners. An important future direction will be preparing accessible guidance materials that address theoretical, methodological, and pragmatic aspects of applying *Certus* to real-world assurance cases.

Bibliography

[1] U.S. National Transportation Safety Board, "Collision Between Vehicle Controlled by Developmental Automated Driving System and Pedestrian," U.S. National Transportation Safety Board, Accident Report NTSB/HAR-19/03, Nov. 2019.

[2] P. Koopman, "Anatomy of a Robotaxi Crash: Lessons from the Cruise Pedestrian Dragging Mishap," in *Computer Safety, Reliability, and Security (SAFECOMP) 2024*, ser. Lecture Notes in Computer and Science, vol. 14988. Springer, 2024, pp. 119–133.

[3] M. Cummings and B. Bauchwitz, "Identifying Research Gaps through Self-Driving Car Data Analysis," *IEEE Transactions on Intelligent Vehicles*, pp. 1–10, 2024.

[4] N. Leveson and C. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993.

[5] "Ariane 501 - Flight 501 Failure - Report by the Inquiry Board," Ariane 501 Inquiry Board, Paris, France, Tech. Rep., Jul. 1996.

[6] "Patriot Missile Defense - Software Problem Led to System Failure at Dhahran, Saudi Arabia," United States General Accounting Office, Washington, DC, Tech. Rep. GOA/IMTEC-92-26, Feb. 1992.

[7] M. Vancouver, "Metro Vancouver Releases Preliminary Review Findings of Cleveland Dam Event of October 1, 2020," Oct. 2020.

[8] "Summary of the FAA's Review of the Boeing 737 MAX," Federal Aviation Administration, Tech. Rep., Nov. 2020.

[9] "IEEE 1471 - Recommended Practice for Architectural Description for Software-Intensive Systems," Institute for Electrical and Electronics Engineers, Tech. Rep., Oct. 2000.

[10] S. Diemert, J. H. Weber, and D. Rankin, "Professional Practice Guidelines on Software Development for Safety-Critical Systems," Engineers and Geoscientists of British Columbia, Burnaby, BC, Guideline, 2020.

[11] N. G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety.* Cambridge, Mass: The MIT Press, Jan. 2012.

[12] S. Buchanan, D. Neumann, and J. H. Weber, "Professional Practice Guidelines on Development of Security-Critical Software," Engineers and Geoscientists of British Columbia, Burnaby, BC, Guideline, 2025.

[13] "Road vehicles - Functional safety," International Organization for Standardization, Standard ISO 26262, 2018.

[14] "RTCA DO-178C - Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, Washington, DC, Standard, 2011.

[15] "EN 50128 - Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems," CENELEC, Standard, 2011.

[16] A. Wassyng, T. Maibaum, M. Lawford, and H. Bherer, "Software Certification: Is There a Case against Safety Cases?" in *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, no. 6662, pp. 206–227.

[17] J. Birch, R. Rivett, I. Habli, B. Bradshaw, J. Botham, D. Higham, P. Jesty, H. Monkhouse, and R. Palin, "Safety Cases and Their Role in ISO 26262 Functional Safety Assessment," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 154–165.

[18] R. Ashmore, R. Calinescu, and C. Paterson, "Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges," *ACM Computing Surveys*, vol. 54, no. 5, pp. 1–39, Jun. 2022.

[19] C. Paterson, R. Hawkins, C. Picardi, Y. Jia, R. Calinescu, and I. Habli, "Safety assurance of Machine Learning for autonomous systems," *Reliability Engineering & System Safety*, vol. 264, p. 111311, Dec. 2025.

[20] Assurance Case Working Group, "Goal Structuring Notation Community Standard (Version 3)," Safety Critical Systems Club, Standard, 2021.

[21] P. Koopman and M. Wagner, "Positive Trust Balance for Self-driving Car Deployment," in *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 351–357.

[22] A. C. W. Group, "Assurance Case Guidance - Challenges, Common Issues and Good Practice (version 1.1)," Safety Critical Systems Club, Guidance, 2021.

[23] "Aurora's Safety Case Framework," Aurora Innovation, Tech. Rep., 2021. [Online]. Available: https://safetycaseframework.aurora.tech/gsn

[24] F. M. Favarò, L. Fraade-Blanar, S. Schnelle, T. Victor, M. Peña, J. Engström, J. Scanlon, K. Kusano, and D. Smith, "Building a credible case for safety: Approach proposal for Automated Driving Systems," *Journal of Safety Research*, vol. 96, pp. 181–196, Feb. 2026.

[25] "ISO/PAS 8800:2024 Road vehicles — Safety and artificial intelligence," International Organization for Standardization, Standard 8800, 2024.

[26] "Structured Assurance Case Metamodel (SCAM)," Object Management Group, Standard, 2021.

[27] T. P. Kelly, "Arguing Safety - A Systematic Approach to Safety Case Management," PhD Thesis, 1998.

[28] P. Bishop and R. Bloomfield, "A Methodology for Safety Case Development," in *Industrial Perspectives on System Safety*. Springer, 1998.

[29] W. D. Cullen, "The Public Inquiry into the Piper Alpha Disaster," United Kingdom Department of Energy, Tech. Rep., Nov. 1990.

[30] S. E. Toulmin, *The Uses of Argument*, 2nd ed. Cambridge University Press, 2003.

[31] J. B. Goodenough, C. B. Weinstock, and A. Z. Klein, "Eliminative Argumentation: A Basis for Arguing Confidence in System Properties," Carnegie Mellon University - Software Engineering Institute, Tech. Rep., 2015.

[32] C. M. Holloway, "The Friendly Argument Notation (FAN): 2023 Version," National Aeronautics and Space Administration (NASA), Tech. Rep. NASA/TM-20230004423, 2023.

[33] "Road vehicles - Cybersecurity engineering," International Organization for Standardization, Standard ISO 21434, 2021.

[34] "Standard for Evaluation of Autonomous Products," Underwritter Laboratories, Standard UL 4600, 2022.

[35] I. T. Committee, "Guidelines and recommendations for ADS safety requirements, assessments and test methods to inform regulatory development," United Nations Economic Commission for Europe, Tech. Rep., 2025.

[36] "REGDOC-1.1.3 - License Application Guide: License to Operate a Nuclear Power Plan," Canadian Nulcear Safety Commission, Regulation, 2016.

[37] Office for Nuclear Regulation, "The Purpose, Scope, and Content of Safety Cases," Government of the United Kingdom, Tech. Rep. NS-TAST-GD-051, 2019.

[38] A. Nack, S. Diemert, and A. Campbell, "First of a Kind Use of an Assurance Case in NEI 20-07 to Address Common Cause Failure." Chicago, United States: The American Nuclear Society (ANS), Jun. 2025.

[39] "Railway Applications - The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) - Part 1: Generic RAMS Process," European Committee for Electrotechnical Standardization, Standard EN 50126, 2017.

[40] A. for the Advancement of Medical Instrumentation, "AAMI TIR38:2019 - Medical Device Safety Assurance Case Guidance," Arlington, USA, 2019.

[41] Z. Daw, S. Beecher, M. Holloway, and M. Graydon, "Overarching Properties as means of compliance: An industrial case study," in *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*, Oct. 2021, pp. 1–10.

[42] U. K. M. of Defence, "Defence Standard 00-056 - Safety Management Requirements for Defence Systems," Standard Def Stan 00-056, 2023.

[43] "Canadian Forces Technical Order - Naval Materiel Safety Management," Canadian Department of National Defence, Ottawa, Canada, Tech. Rep., 2025.

[44] "United Kingdom - The Offshore Installations (Safety Case) Regulations 2005," 2005.

[45] R. Hawkins, "Developing Compelling Safety Cases," Feb. 2025. [Online]. Available: http://arxiv.org/abs/2502.00911

[46] R. Bloomfield and J. Rushby, "Assessing Confidence with Assurance 2.0," SRI International, Tech. Rep. SRI-CSL-2022-02, 2022.

[47] R. Hawkins, T. Kelly, J. Knight, and P. Graydon, "A New Approach to Creating Clear Safety Arguments," in *Advances in Systems Safety*. Springer, 2011, pp. 3–23.

[48] C. Hobbs and M. Lloyd, "The Application of Bayesian Belief Networks to Assurance Case Preparation," in *Achieving Systems Safety*. Springer, 2012, pp. 159–176.

[49] C. M. Holloway and K. S. Wasson, "A Primer on Argument Assessment," https://ntrs.nasa.gov/citations/20210022807, National Aeronautics and Space Administration (NASA) and Joby Aviation, Tech. Rep., 2021.

[50] Y. Idmessaoud, D. Dubois, and J. Guiochet, "Confidence assessment in safety argument structure - Quantitative vs. qualitative approaches," *International Journal of Approximate Reasoning*, vol. 165, p. 109100, 2024.

[51] J. Fenn, R. Hawkins, and M. Nicholson, "A New Approach to Creating Clear Operational Safety Arguments," in *Computer Safety, Reliability, and Security. SafeCOMP 2024 Workshops*. Cham: Springer, 2024, pp. 227–238.

[52] S. Diemert, C. Shortt, and J. H. Weber, "How do practitioners gain confidence in assurance cases?" *Information and Software Technology*, vol. 185, p. 107767, Sep. 2025.

[53] S. Diemert and J. Joyce, "Eliminative Argumentation for Arguing System Safety - A Practitioner's Experience," in *2020 IEEE International Systems Conference (SysCon)*, 2020, pp. 1–7.

[54] N. G. Leveson, "The Use of Safety Cases in Certification and Regulation," https://dspace.mit.edu/handle/1721.1/102833, 2011.

[55] C. Haddon-Cav, "The Nimrod Review," House of Commons of the United Kingdom, Independent Review, 2009.

[56] E. Denney, G. Pai, and I. Habli, "Dynamic Safety Cases for Through-Life Safety Assurance," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 587–590.

[57] S. Diemert, J. Goodenough, J. Joyce, and C. Weinstock, "Incremental assurance through eliminative argumentation," *Journal of System Safety*, vol. 58, no. 1, pp. 7–15, 2023.

[58] C. Hobbs, S. Diemert, and J. Joyce, "Driving the Development Process from the Safety Case," in *Safe AI Systems*. Safety-Critical Systems Club, 2024.

[59] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," *IEEE Software*, vol. 33, no. 3, pp. 94–100, May 2016.

[60] M. Zeller, "Towards Continuous Safety Assessment in Context of DevOps," in *Computer Safety, Reliability, and Security. SAFECOMP 2021 Workshops*. Springer, Cham, 2021, pp. 145–157.

[61] A. Nouri, B. Cabrero-Daniel, F. Törner, and C. Berger, "The DevSafeOps dilemma: A systematic literature review on rapidity in safe autonomous driving development and operation," *Journal of Systems and Software*, vol. 230, p. 112555, Dec. 2025.

[62] P. Schleiss, F. Carella, and I. Kurzidem, "Towards Continuous Safety Assurance for Autonomous Systems," in *2022 6th International Conference on System Reliability and Safety (ICSRS)*, Nov. 2022, pp. 457–462.

[63] E. Asaadi, E. Denney, J. Menzies, G. J. Pai, and D. Petroff, "Dynamic Assurance Cases: A Pathway to Trusted Autonomy," *Computer*, vol. 53, no. 12, pp. 35–46, 2020.

[64] E. Denney and G. Pai, "Reconciling Safety Measurement and Dynamic Assurance," in *Computer Safety, Reliability, and Security.* Cham: Springer Nature Switzerland, 2024, pp. 51–67.

[65] M. Zeller, T. Waschulzik, R. Schmid, and C. Bahlmann, "Toward a safe MLOps process for the continuous development and safety assurance of ML-based systems in the railway domain," *AI and Ethics*, vol. 4, no. 1, pp. 123–130, Feb. 2024.

[66] M. Zeller, T. Waschulzik, C. Carlan, M. Serahlazau, C. Bahlmann, Z. Wu, S. Spieckermann, D. Krompass, S. Geerkens, C. Siebericchs, K. Kirchheim, B. K. Özen, and L. D. Robles, "Continuous Development and Safety Assurance Pipeline for ML-Based Systems in the Railway Domain," in *Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops.* Cham: Springer Nature Switzerland, 2024, pp. 446–459.

[67] D. Weyns, *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective.* John Wiley & Sons, Oct. 2020.

[68] R. de Lemos, D. Garlan, C. Ghezzi, H. Giese, J. Andersson, M. Litoiu, B. Schmerl, D. Weyns, L. Baresi, N. Bencomo, Y. Brun, J. Camara, R. Calinescu, M. B. Cohen, A. Gorla, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, R. Mirandola, M. Mori, H. A. Müller, R. Rouvoy, C. M. F. Rubira, E. Rutten, M. Shaw, G. Tamburrelli, G. Tamura, N. M. Villegas, T. Vogel, and F. Zambonelli, "Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances," in *Software Engineering for Self-Adaptive Systems III. Assurances*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 3–30.

[69] S. Jahan, "An Adaptation Assessment Framework for Runtime Security Assurance Case Evolution," Ph.D., The University of Tulsa, Oklahoma, USA, 2021.

[70] S. Diemert and J. Weber, "Safety-Critical Adaptation in Self-Adaptive Systems," in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2022, pp. 371–380.

[71] S. Diemert and J. H. Weber, "Hazard Analysis for Self-Adaptive Systems Using System-Theoretic Process Analysis," in *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2023, pp. 145–156.

[72] D. Schneider, "Conditional Safety Certification for Open Adaptive Systems," Dissertation, Technical University of Kaiserslautern, Fraunhofer IESE, 2014.

[73] E. Mirzaei, C. Thomas, and M. Conrad, "Safety Cases for Adaptive Systems of Systems: State of the Art and Current Challenges," in *Dependable Computing - EDCC 2020 Workshops*, ser. Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 127–138.

[74] C. Thomas, E. Mirzaei, B. Wudka, L. Siefke, and V. Sommer, "Service-Oriented Reconfiguration in Systems of Systems Assured by Dynamic Modular Safety Cases," in *17th European Dependable Computing Conference, EDCC 2021*, ser. Communications in Computer and Information Science, vol. 1462. Munich, Germany: Springer Science and Business Media Deutschland GmbH, 2021, pp. 12–29.

[75] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly, "Engineering trustworthy self-adaptive software with dynamic assurance cases," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1039–1069, 2018.

[76] J. Cheng, M. Goodrum, R. Metoyer, and J. Cleland-Huang, "How do practitioners perceive assurance cases in safety-critical software systems?" in *11th International Workshop on Cooperative and Human Aspects of Software Engineering*. Association for Computing Machinery, 2018, pp. 57–60.

[77] L. F. Rivera, H. A. Müller, N. M. Villegas, G. Tamura, and M. Jiménez, "On the Engineering of IoT-Intensive Digital Twin Software Systems," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ser. ICSEW'20. New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 631–638.

[78] S. Nair, J. L. de la Vara, M. Sabetzadeh, and D. Falessi, "Evidence management for compliance of critical systems with safety standards: A survey on the state of practice," *Information and Software Technology*, vol. 60, pp. 1–15, 2015.

[79] L. Cyra and J. Górski, "Support for argument structures review and assessment," *Reliability Engineering & System Safety*, vol. 96, no. 1, pp. 26–37, 2011.

[80] P. J. Graydon and C. M. Holloway, "An investigation of proposed techniques for quantifying confidence in assurance arguments," *Safety Science*, vol. 92, pp. 53–65, 2017.

[81] J. B. Goodenough, C. B. Weinstock, and A. Z. Klein, "Eliminative Induction: A basis for arguing system confidence," in *35th International Conference on Software Engineering*, 2013, pp. 1161–1164.

[82] P. J. Graydon, "Defining baconian probability for use in assurance argumentation," National Aeronautics and Space Adminstration (NASA), Tech. Rep. NASA/TM-2016-219341, 2016.

[83] B. Glaser and A. Strauss, *Discovery of Grounded Theory: Strategies for Qualitative Research*, 1st ed. Routledge, 1999.

[84] S. Diemert, L. Millet, J. Joyce, and J. H. Weber, "Including Defeaters in Quantitative Confidence Assessments for Assurance Cases," in *Computer Safety, Reliability, and Security. SafeCOMP 2024 Workshops*, ser. Lecture Notes in Computer Science, vol. 14989. Springer, 2024, pp. 239–250.

[85] S. Diemert and J. H. Weber, "Certus: A Domain Specific Language for Confidence Assessment in Assurance Cases," in *Computer Safety, Reliability, and Security. SAFECOMP 2025 Workshops*. Cham: Springer Nature Switzerland, 2025, pp. 211–225.

[86] "Guidelines for Automotive Safety Arguments," Motor Industry Software Reliability Association (MISRA), Tech. Rep., 2019.

[87] I. Habli, R. Hawkins, C. Paterson, P. Ryan, Y. Jia, M. Sujan, and J. McDermid, "The BIG Argument for AI Safety Cases," Mar. 2025. [Online]. Available: http://arxiv.org/abs/2503.11705

[88] P. J. Graydon, J. C. Knight, and E. A. Strunk, "Assurance Based Development of Critical Systems," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, Jun. 2007, pp. 347–357, iSSN: 2158-3927.

[89] Oxford English Dictionary, "dialectic, n.[1], sense 1.a," Mar. 2025. [Online]. Available: https://doi.org/10.1093/OED/1078772731

[90] S. Kokaly, "Managing Assurance Cases in Model Based Software Systems," Dissertation, McMaster University, Hamilton, Ontario, 2019.

[91] T. Viger, J. Joyce, S. Diemert, C. Menghi, M. Chechik, J. Uythoven, M. Zerlauth, and L. Felsberfer, "Assessing the Usefulness of Assurance Cases: Experience with the Large Hadron Collider," *Systems Engineering*, 2025.

[92] P. Koopman, *How Safe Is Safe Enough?: Measuring and Predicting Autonomous Vehicle Safety*. Pittsburgh, Pennsylvania: Independently published, Sep. 2022.

[93] E. C. Wiebe and A. Weaver, "The Vancouver Island School-Based Weather Station Network," in *American Meteorological Society Meeting Abstracts*, vol. 97, 2017.

[94] T. P. Kelly and J. A. McDermid, "Safety Case Construction and Reuse Using Patterns," in *Safe Comp 97*. Springer, London, 1997, pp. 55–69.

[95] I. Habli, "Model-Based Assurance of Safety-Critical Product Lines," Ph.D. dissertation, University of York, York, UK, 2009.

[96] I. Habli and T. Kelly, "A Safety Case Approach to Assuring Configurable Architectures of Safety-Critical Product Lines," in *Architecting Critical Systems*. Berlin, Heidelberg: Springer, 2010, pp. 142–160.

[97] T. Chowdhury, C.-W. Lin, B. Kim, M. Lawford, S. Shiraishi, and A. Wassyng, "Principles for Systematic Development of an Assurance Case Template from ISO 26262," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2017, pp. 69–72.

[98] S. Kokaly, R. Salay, V. Cassano, T. Maibaum, and M. Chechik, "A model management approach for assurance case reuse due to system evolution," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 196–206.

[99] N. Annable, T. Chiang, M. Lawford, R. F. Paige, and A. Wassyng, "Generating Assurance Cases Using Workflow+ Models," in *Computer Safety, Reliability, and Security*, M. Trapp, F. Saglietti, M. Spisländer, and F. Bitsch, Eds. Cham: Springer International Publishing, 2022, pp. 97–110.

[100] N. Annable, M. Lawford, R. F. Paige, and A. Wassyng, "Generating Understandable and Reusable Safety Assurance Cases using Workflow+," in *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2024, pp. 236–239.

[101] N. Annable, M. Askarpour, T. Chiang, S. Kokaly, M. Lawford, R. F. Paige, R. Sethu, and A. Wassyng, "Comprehensive Change Impact Analysis Applied to Advanced Automotive Systems," in *Computer Safety, Reliability, and Security*, A. Ceccarelli, M. Trapp, A. Bondavalli, and F. Bitsch, Eds. Cham: Springer Nature Switzerland, 2024, pp. 134–149.

[102] M. Castillo-Effen, C. E. Veldhuizen, and C. D. Lutz, "A Digital Assurance Framework," in *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2024, pp. 232–235.

[103] T. E. Wang, C. Oh, M. Low, I. Amundson, Z. Daw, A. Pinto, M. L. Chiodo, G. Wang, S. Hasan, R. Melville, and P. Nuzzo, "Computer-Aided Generation of Assurance Cases," in *Computer Safety, Reliability, and Security. SAFECOMP 2023 Workshops*, J. Guiochet, S. Tonetta, E. Schoitsch, M. Roy, and F. Bitsch, Eds. Cham: Springer Nature Switzerland, 2023, pp. 135–148.

[104] S. Paul, B. Meng, K. Siu, A. Moitra, and M. Durling, "Assurance Case Synthesis from a Curated Semantic Triplestore," in *Computer Safety, Reliability, and Security*. Cham: Springer Nature Switzerland, 2024, pp. 84–99.

[105] D. Nešić, M. Nyberg, and B. Gallina, "Product-line assurance cases from contract-based design," *Journal of Systems and Software*, vol. 176, p. 110922, Jun. 2021.

[106] L. Murphy, T. Viger, A. D. Sandro, and M. Chechik, "PLACIDUS: Engineering Product Lines of Rigorous Assurance Cases," in *Integrated Formal Methods*. Cham: Springer Nature Switzerland, 2025, pp. 87–108.

[107] M. Sivakumar, A. B. Belle, J. Shan, and K. Khakzad Shahandashti, "Prompting GPT-4 to support automatic safety case generation," *Expert Systems with Applications*, vol. 255, p. 124653, Dec. 2024.

[108] O. Odu, A. B. Belle, S. Wang, S. Kpodjedo, T. C. Lethbridge, and H. Hemmati, "Automatic instantiation of assurance cases from patterns using large language models," *Journal of Systems and Software*, vol. 222, p. 112353, Apr. 2025.

[109] O. Odu, D. M. Beltrán, E. B. Gutiérrez, A. B. Belle, G. Yu, and M. Sherafat, "SmartGSN: An Online Tool to Semi-automatically Manage Assurance Cases," in *Computer Safety, Reliability, and Security*. Cham: Springer Nature Switzerland, 2026.

[110] Z. Chen, Y. Deng, and W. Du, "Trusta: Reasoning about assurance cases with formal methods and large language models," *Science of Computer Programming*, vol. 244, p. 103288, Sep. 2025.

[111] S. Diemert, E. Cyffka, N. Anwari, O. Foster, T. Viger, L. Millet, and J. Joyce, "Balancing the Risks and Benefits of Using Large Language Models to Support Assurance Case Development," in *Computer Safety, Reliability, and Security*. Cham: Springer Nature Switzerland, 2025, pp. 209–225.

[112] M. S. Graydon and S. M. Lehman, "Examining Proposed Uses of LLMs to Produce or Assess Assurance Arguments," Tech. Rep., Mar. 2025. [Online]. Available: https://ntrs.nasa.gov/citations/20250001849

[113] S. Diemert, T. Viger, J. Joyce, and M. Chechik, "One-size-fits-all Evaluation of LLMs for Safety Assurance Considered Harmful," in *SAFECOMP 2025 Position Papers*, Sep. 2025. [Online]. Available: https://laas.hal.science/hal-05241986

[114] K. K. Shahandashti, A. B. Belle, T. C. Lethbridge, O. Odu, and M. Sivakumar, "A PRISMA-driven systematic mapping study on system assurance weakeners," *Information and Software Technology*, vol. 175, p. 107526, 2024.

[115] C. Hobbs, "Experience with Assurance Case Preparation," BlackBerry QNX, Experience Report, 2019.

[116] L.-P. Cobos, T. Miao, K. Sowka, G. Madzudzo, A. R. Ruddle, and E. El Amam, "Application of an Automotive Assurance Case Approach to Autonomous Marine Vessel Security," in *2022 International Conference on Electrical, Computer,*

Communications and Mechatronics Engineering (ICECCME), Nov. 2022, pp. 1–9.

[117] R. Hawkins and P. Ryan Conmy, "Identifying Run-Time Monitoring Requirements for Autonomous Systems Through the Analysis of Safety Arguments," in *Computer Safety, Reliability, and Security.* Cham: Springer Nature Switzerland, 2023, pp. 11–24.

[118] P. Ryan, S. Shahbeigi, J. Zou, I. Stefanakos, and J. Molloy, "A Dynamic Assurance Framework for an Autonomous Survey Drone," in *Computer Safety, Reliability, and Security. SAFECOMP 2024.* Springer, 2024, pp. 285–299.

[119] J. B. Goodenough, C. B. Weinstock, and A. Z. Klein, "Toward a Theory of Assurance Case Confidence," Carnegie Mellon University - Software Engineering Institute, Tech. Rep. CMU/SEI-2012-TR-002, Sep. 2012. [Online]. Available: https://apps.dtic.mil/docs/citations/ADA609836

[120] Oxford English Dictionary, "defeasible, adj., sense 2," Sep. 2025. [Online]. Available: https://doi.org/10.1093/OED/3159143288

[121] T. Viger, L. Murphy, S. Diemert, C. Menghi, A. Di, and M. Chechik, "Supporting Assurance Case Development Using Generative AI," in *SAFECOMP 2023, Position Papers*, Toulouse, France, Sep. 2023.

[122] T. Viger, L. Murphy, S. Diemert, C. Menghi, J. Joyce, A. Di Sandro, and M. Chechik, "AI-Supported Eliminative Argumentation: Practical Experience Generating Defeaters to Increase Confidence in Assurance Cases," in *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2024, pp. 284–294.

[123] U. Gohar, M. C. Hunter, R. R. Lutz, and M. B. Cohen, "CoDefeater: Using LLMs To Find Defeaters in Assurance Cases," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, pp. 2262–2267.

[124] K. K. Shahandashti, A. B. Belle, M. M. Mohajer, O. Odu, T. C. Lethbridge, H. Hemmati, and S. Wang, "Using GPT-4 Turbo to Automatically Identify

Defeaters in Assurance Cases," in *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, Jun. 2024, pp. 46–56.

[125] K. S. Wasson and C. M. Holloway, "An Introduction to Constructing and Assessing Overarching Properties Related Arguments (OPRAs): Version 1.0," Joby Aviation and National Aeronautics and Space Administration (NASA), Tech. Rep., 2022.

[126] T. Chowdhury, A. Wassyng, R. F. Paige, and M. Lawford, "Criteria to Systematically Evaluate (Safety) Assurance Cases," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2019, pp. 380–390.

[127] T. Chowdhury, A. Wassyng, R. Paige, and M. Lawford, "Systematic Evaluation of (Safety) Assurance Cases," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 18–33.

[128] R. Weaver, J. Fenn, and T. Kelly, "A pragmatic approach to reasoning about the assurance of safety arguments," in *Proceedings of the 8th Australian workshop on Safety critical systems and software*, vol. Volume 33, 2003, pp. 57–67.

[129] A. Hafver, D. McGeorge, F. Pedersen, and R. Flage, "Reconsidering confidence in assurance cases: From quantification to strength-of-knowledge aggregation," in *SAFECOMP 2025, Position Papers*, Stockholm, Sweden, 2025.

[130] A. Hafver, D. McGeorge, E. Stensrud, F. Pedersen, T. Skramstad, and R. Flage, "Propagating knowledge strength through assurance arguments using three-valued logic to assess confidence in claims," in *Proceedings of the 35th European Safety and Reliability & the 33rd Society for Risk Analysis Europe Conference*. Research Publishing, Singapore, 2025.

[131] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, 1st ed. San Francisco, Calif: Morgan Kaufmann, Sep. 1988.

[132] N. E. Fenton and N. Martin, *Risk Assessment and Decision Analysis with Bayesian Networks*, second edition ed. Boca Raton, FL: CRC Press, 2019.

[133] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. New Jersey, USA: Pearson Education Inc., 2010.

[134] N. Fenton, B. Littlewood, M. Neil, L. Strigini, A. Sutcliffe, and D. Wright, "Assessing dependability of safety critical systems using diverse evidence," *IEE Proceedings - Software*, vol. 145, no. 1, pp. 35–39, Feb. 1998.

[135] B. Littlewood, L. Strigini, D. Wright, and P. Courtois, "Examination of Bayesian belief network for safety assessment of nuclear computer-based systems," DeVa ESPRIT Long Term Research Project, Tech. Rep., 1998.

[136] B. Guo, "Knowledge representation and uncertainty management: applying Bayesian belief networks to a safety assessment expert system," in *International Conference on Natural Language Processing and Knowledge Engineering, 2003. Proceedings. 2003*, Oct. 2003, pp. 114–119.

[137] X. Zhao, D. Zhang, M. Lu, and F. Zeng, "A New approach to Assessment of Confidence in Assurance Cases," in *International Conference on Computer Safety, Reliability and Security (SafeCOMP) 2012*, ser. Lecture Notes in Computer Science, vol. 7613. Springer, 2012, pp. 79–91.

[138] D. Nešić, M. Nyberg, and B. Gallina, "A probabilistic model of belief in safety cases," *Safety Science*, vol. 138, p. 105187, Jun. 2021.

[139] J. Rushby, "On the Interpretation of Assurance Case Arguments," in *New Frontiers in Artificial Intelligence*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 331–347.

[140] C. Oh, N. Naik, Z. Daw, T. E. Wang, and P. Nuzzo, "ARACHNE: Automated Validation of Assurance Cases with Stochastic Contract Networks," in *International Conference on Computer Safety, Reliability and Security, SafeCOMP 2022*, ser. Lecture Notes in Computer Science, vol. 13414. Springer, 2022, pp. 65–81.

[141] G. Shafer, *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[142] J. Halpern, Y., *Reasoning About Uncertainty*. Cambridge, Mass: MIT Press, 2003.

[143] T. J. Ross, *Fuzzy Logic with Engineering Applications, Third Edition*, 3rd ed. Chichester, U.K: Wiley, Mar. 2010.

[144] R. Wang, J. Guiochet, G. Motet, and W. Schön, "Safety case confidence propagation based on Dempster-Shafer theory," *International Journal of Approximate Reasoning*, vol. 107, pp. 46–64, Apr. 2019.

[145] L. Cyra and J. Górski, "An Approach to Evaluation of Arguments in Trust Cases," in *2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, Jun. 2008, pp. 103–110.

[146] L. Cyra and J. Górski, "Expert Assessment of Arguments: A Method and Its Experimental Evaluation," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 291–304.

[147] A. Ayoub, J. Chang, O. Sokolsky, and I. Lee, "Assessing the Overall Sufficiency of Safety Arguments," in *21st Safety-Critical Systems Symposium (SSS'13)*. Safety-Critical Systems Club, 2013, pp. 127–144.

[148] R. Wang, "Confidence in safety argument - An assessment framework based on belief function theory," Dissertation, L'Universite Federale Toulouse, Toulouse, France, 2018.

[149] R. Wang, J. Guiochet, G. Motet, and W. Schön, "Modelling confidence in railway safety case," *Safety Science*, vol. 110, pp. 286–299, Dec. 2018.

[150] J. Guiochet, Q. A. Do Hoang, and M. Kaaniche, "A model for safety case confidence assessment," in *34th International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2015, September 23, 2015 - September 25, 2015*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9337. Delft, Netherlands: Springer Verlag, 2015, pp. 313–327.

[151] A. Jøsang, *Subjective Logic*, ser. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016.

[152] L. Duan, S. Rayadurgam, M. P. E. Heimdahl, O. Sokolsky, and I. Lee, "Representing Confidence in Assurance Case Evidence," in *Computer Safety, Reliability, and Security (SAFECOMP 2015)*, ser. Lecture Notes in Computer Science, vol. 9338. Springer, 2015, pp. 15–26.

[153] C. Yuan, J. Wu, C. Liu, and a. H. Yang, "A Subjective Logic-Based Approach for Assessing Confidence in Assurance Case," *International Journal of Performability Engineering*, vol. 13, no. 6, p. 807, 2017.

[154] B. Herd, J.-V. Zacchi, and S. Burton, "A Deductive Approach to Safety Assurance: Formalising Safety Contracts with Subjective Logic," in *Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops*, ser. Lecture Notes in Computer Science, vol. 14989. Springer, 2024, pp. 213–226.

[155] S. Jahan, A. Marshall, and R. F. Gamble, "Evaluating security assurance case adaptation," in *52nd Annual Hawaii International Conference on System Sciences, HICSS 2019, January 8, 2019 - January 11, 2019*, ser. Proceedings of the Annual Hawaii International Conference on System Sciences, vol. 2019-January. Maui, HI, United states: IEEE Computer Society, 2019, pp. 7312–7321.

[156] R. Bloomfield and J. Rushby, "Assurance 2.0: A Manifesto," Jan. 2021.

[157] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[158] B. H. C. Cheng, R. J. Clark, J. E. Fleck, M. A. Langford, and P. K. McKinley, "AC-ROS: Assurance Case Driven Adaptation for the Robot Operating System," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 102–113.

[159] E. Asaadi, E. Denney, and G. Pai, "Quantifying Assurance in Learning-Enabled Systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, A. Casimiro, F. Ortmeier, F. Bitsch, and P. Ferreira, Eds. Cham: Springer International Publishing, 2020, pp. 270–286.

[160] U. D. Ferrell and A. H. A. Anderegg, "Validation of Assurance Case for Dynamic Systems," in *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*, Sep. 2022, pp. 1–11.

[161] S. Jahan, M. Pasco, R. Gamble, P. McKinley, and B. Cheng, "MAPE-SAC: A Framework to Dynamically Manage Security Assurance Cases," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Jun. 2019, pp. 146–151.

[162] S. Jahan, I. Riley, C. Walter, R. F. Gamble, M. Pasco, P. K. McKinley, and B. H. C. Cheng, "MAPE-K/MAPE-SAC: An interaction framework for adaptive systems with security assurance cases," *Future Generation Computer Systems*, vol. 109, pp. 197–209, Aug. 2020.

[163] D. Schneider and M. Trapp, "A Safety Engineering Framework for Open Adaptive Systems," in *2011 IEEE Fifth International Conference on Self-Adaptive and Self-Organizing Systems*, Oct. 2011, pp. 89–98.

[164] D. Schneider and M. Trapp, "Conditional Safety Certification of Open Adaptive Systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 8, no. 2, pp. 8:1–8:20, Jul. 2013.

[165] "IEC61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems," International Electrotechnical Commission, Geneva, Tech. Rep. IEC61508, 2010.

[166] C. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[167] M. Kasunic, "Designing an Effective Survey," Carnegie Mellon University - Software Engineering Institute, Tech. Rep. CMU/SEI-2005-HB-004, 2005.

[168] J. Linaker, S. M. Sulaman, R. Maiani de Mello, and M. Host, "Guidelines for Conducting Surveys in Software Engineering," Lunds University, Guideline, 2015.

[169] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, Inc., 2019.

[170] R. Passonneau, "Measuring Agreement on Set-valued Items (MASI) for Semantic and Pragmatic Annotation," Columbia University, New York, NY, USA, Tech. Rep., 2006.

[171] K. Aldiabat and C.-L. L. Navenec, "Data Saturation: The Mysterious Step In Grounded Theory Method," *The Qualitative Report*, vol. 23, no. 1, pp. 245–261, Jan. 2018.

[172] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Proceedings of the 38th*

International Conference on Software Engineering, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 120–131.

[173] C. M. Squire, K. C. Giombi, D. J. Rupert, J. Amoozegar, and P. Williams, "Determining an Appropriate Sample Size for Qualitative Interviews to Achieve True and Near Code Saturation: Secondary Analysis of Data," *Journal of Medical Internet Research*, vol. 26, no. 1, p. e52998, Jul. 2024.

[174] L. Millet, S. Diemert, C. Rees, T. Viger, M. Chechik, C. Menghi, and J. Joyce, "Assurance Case Arguments in the Large: The CERN LHC Machine Protection System," in *Computer Safety, Reliability, and Security. SAFECOMP 2023*, ser. Lecture Notes in Computer Science, vol. 14181. Springer, 2023, pp. 3–10.

[175] F. Törner and P. Öhman, "Automotive Safety Case A Qualitative Case Study of Drivers, Usages, and Issues," in *11th IEEE High Assurance Systems Engineering Symposium (HASE)*, 2008, pp. 313–322.

[176] O. Doss and T. P. Kelly, "Challenges and Opportunities in Agile Development in Safety Critical Systems: A Survey," *ACM SIGSOFT Software Engineering Notes*, vol. 41, no. 2, pp. 30–31, 2016.

[177] C. Almendra, C. Silva, L. E. G. Martins, and J. Marques, "How assurance case development and requirements engineering interplay: a study with practitioners," *Requirements Engineering*, vol. 27, pp. 273–292, 2022.

[178] B. Amir and P. Ralph, "There is no random sampling in software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. Association for Computing Machinery, 2018, pp. 344–345.

[179] J. H. Jahnke and A. Walenstein, "Evaluating theories for managing imperfect knowledge in human-centric database reengineering environments," *International Journal of Software Engineering and Knowledge Engineering*, vol. 12, no. 01, pp. 77–102, Feb. 2002.

[180] P. Barbrook-Johnson and A. S. Penn, "Bayesian Belief Networks," in *Systems Mapping: How to build and use causal models of systems*. Cham: Springer International Publishing, 2022, pp. 97–112.

[181] J. Pearl, "Introduction to Probabilities, Graphs, and Causal Models," in *Causality*, 2nd ed. Cambridge: Cambridge University Press, 2009, pp. 1–40.

[182] J. H. Jahnke, "Management of Uncertainty and Inconsistency in Database Reengineering Processes," Dissertation, University of Paderborn, Paderborn, Germany, 1999.

[183] L. Zadeh, "Fuzzy Sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[184] L. A. Zadeh, "Fuzzy logic," *Computer*, vol. 21, no. 4, pp. 83–93, Apr. 1988.

[185] G. Bortolan and R. Degani, "A review of some methods for ranking fuzzy subsets," *Fuzzy Sets and Systems*, vol. 15, no. 1, pp. 1–19, Feb. 1985.

[186] R. R. Yager, "A procedure for ordering fuzzy subsets of the unit interval," *Information Sciences*, vol. 24, no. 2, pp. 143–161, Jul. 1981.

[187] L. A. Zadeh, "Fuzzy sets as a basis for a theory of possibility," *Fuzzy Sets and Systems*, vol. 1, no. 1, pp. 3–28, Jan. 1978.

[188] D. Dubois and H. Prade, *Possibility Theory: An Approach to Computerized Processing of Uncertainty*. New York, NY, USA: Plenum Press, 1988.

[189] D. Dubois, J. Lang, and H. Prade, "Possibilistic logic," Institut de Recherche en Informatique de Toulouse (I.R.I.T), Toulouse, France, Tech. Rep., 1994.

[190] D. Dubois and H. Prade, "Possibilistic Logic — An Overview," in *Handbook of the History of Logic*, ser. Computational Logic. North-Holland, Jan. 2014, vol. 9, pp. 283–342.

[191] R. W. Sebesta, *Concepts of Programming Languages*, 4th ed. Addison-Wesley, 1999.

[192] D. A. Schmidt, *Denotational Semantics: A Methodology for Language Development*. Dubuque, Iowa: William C Brown Pub, Jun. 1988.

[193] S. Diemert, "A mathematical basis for medication prescriptions and adherence," M.Sc. Thesis, University of Victoria, 2017.

[194] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, Jun. 1968.

[195] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[196] J.-F. Monin, *Understanding Formal Methods*. London ; New York: Springer, Jan. 2003.

[197] M. Gordon, J.C., *The Denotational Description of Programming Languages - An Introduction*. New York: Springer-Verlag, 1979.

[198] J. M. Spivey, *The Z Notation: A reference manual*, 2nd ed. Prentice Hall, 1998.

[199] L. Lamport, *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Boston: Addison-Wesley, 2003.

[200] D. Jackson, "Alloy: a language and tool for exploring software designs," *Communications of the ACM*, vol. 62, no. 9, pp. 66–76, Aug. 2019.

[201] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, Mar. 2000.

[202] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[203] L. d. Moura and S. Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Automated Deduction – CADE 28*. Cham: Springer International Publishing, 2021, pp. 625–635.

[204] T. mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, Jan. 2020, pp. 367–381.

[205] S. Ullrich and L. de Moura, "Beyond Notations: Hygienic Macro Expansion for Theorem Proving Languages," in *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, Jul. 2020, pp. 167–182.

[206] A. Paulino, D. Testa, E. Ayers, E. Karunus, H. Boving, J. Limperg, S. Gadgil, and S. Bhat, "Metaprogramming in Lean 4," Tech. Rep., 2025. [Online]. Available: https://leanprover-community.github.io/lean4-metaprogramming-book/

[207] C. Carlan, "Checkable Safety Arguments-A Modeling Framework Supporting the Maintenance of Safety Arguments Consistent with System Development Artifacts," Dissertation, Technische Universität München, Munich, Germany, 2025.

[208] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, Sep. 2014.

[209] A. Casey, L. Millet, J. Joyce, V. Nachiappan, and S. Keane, "Using Eliminative Argumentation to Enhance Trust in ILI Results," in *Proceedings of the ASME 2024 14th International Pipeline Conference (IPC2024)*, Calgary, Canada, Feb.

[210] "Demonstration Safety Case Argument for the Gravity Ventilator (gVent)," Critical Systems Labs Inc., Canada, Tech. Rep., 2025. [Online]. Available: https://criticalsystemslabs.com/wp-content/uploads/2025/12/gVent-Safety-Case.pdf

[211] M. Zeroual, B. Hamid, M. Adedjouma, and J. Jaskolka, "Formal model-based argument patterns for security cases," in *Proceedings of the 28th European Conference on Pattern Languages of Programs*, ser. EuroPLoP '23. New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 1–12.

[212] C. Hartsell, N. Mahadevan, A. Dubey, and G. Karsai, "Automated Method for Assurance Case Construction from System Design Models," in *2021 5th International Conference on System Reliability and Safety (ICSRS)*, Nov. 2021, pp. 230–239.

[213] M. Vierhauser, S. Bayley, J. Wyngaard, J. Cheng, W. Xiong, R. Lutz, J. Huseman, and J. Cleland-Huang, "Interlocking Safety Cases for Unmanned Autonomous Systems in Urban Environments," in *Proceedings of the 40th Inter-*

national Conference on Software Engineering: Companion Proceeedings, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 416–417.

[214] F. R. Ward and I. Habli, "An Assurance Case Pattern for the Interpretability of Machine Learning in Safety-Critical Systems," in *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*. Cham: Springer International Publishing, 2020, pp. 395–407.

[215] C.-L. Lin, W. Shen, and R. Hawkins, "Support for safety case generation via model transformation," *SIGBED Rev.*, vol. 14, no. 2, pp. 44–52, Mar. 2017.

[216] B. Xu, M. Lu, and D. Zhang, "A Layered Argument Strategy for Software Security Case Development," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2017, pp. 331–338.

[217] "Rapidly Manufactured Ventilator System (RMVS)," Medicines & Healthcare Products Regulatory Agency of the United Kingdom, United Kingdom, Tech. Rep. RMVS001-v4, 2020.

[218] M. Host, A. Rainer, P. Runeson, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*, 1st ed. Hoboken, N.J: Wiley, Apr. 2012.

[219] C. Carlan, "Checkable Safety Arguments-A Modeling Framework Supporting the Maintenance of Safety Arguments Consistent with System Development Artifacts," Dissertation, Technische Universität München, Munich, Germany, 2025.

Appendix A
Artifacts from Practitioner Interviews

This appendix contains instruments and detailed results from the practitioner interviews presented in Chapter 3. These artifacts were published in [52].

A.1 Interview Instrument

The interview of practitioners consisted of the following questions:

1. [**SCREENING**] How many years of professional experience do you have? Both in general and working in systems or software assurance? *Participant must report some amount of professional experience.*

2. What industry or industries do you have experience working in? What industries have you prepared ACs in?

3. [**SCREENING**] Please describe your role as it relates to the preparation or development of ACs.

4. What are the quality attributes being assured in the ACs you have worked on? For example, a "safety case".

5. How are the ACs that you have worked on expressed or captured?

6. How have you evaluated or assessed your degree of belief or confidence in an AC? What methods, techniques, or activities have you used?

7. Have you ever used a quantitative method for assessing confidence as part of a real-world project? If so, what was your opinion of the method and the outcome? (Follow-up: If yes, what methods have you used?).

8. Given that you have performed some type of confidence assessment, how have you used the results?

9. [**MULTIPLE CHOICE**] What impact did the methods, techniques, or activities you used to assess confidence have on the ACs you have worked on? Select from: A) Positive Impact, B) Neutral Impact, or C) Negative Impact

10. What challenges or barriers do you think exist, or have you experienced, to conducting confidence assessments for ACs in practice?

11. [**MULTIPLE CHOICE**] Regardless of what you or your organization have done in the past, do you believe a dedicated confidence assessment activity is important? Should it be: A) Mandatory, B) Recommended, C) Optional, D) Discouraged

A.2 Email Questionnaire

The follow-up email questionnaire consisted of the following questions:

1. In your industrial or practical experience, what role have models of systems (e.g., SysML models) had informing your confidence in an assurance case? For example, have you used models as evidence, to generate assurance arguments, or perform tasks related to developing or maintaining an assurance case? If so, what models, methods, or tools were used, and how did they impact your confidence in the assurance case?

2. How do you imagine (regardless of your experience to date), model-based approaches might be used to inform confidence assessment of assurance cases? That is, do you think model-based approaches could be useful for this purpose? If yes, how so? If not, why?

3. In your industrial or practical experience, have you used formal methods (e.g., theorem provers or model checkers) to analyze an assurance case for correctness or completeness? If so, what formal methods or tools were used, and how did they impact your confidence in the assurance case?

4. How do you imagine (regardless of your experience to date), formal methods might be used to inform confidence assessment of assurance cases? That is, do

you think formal methods could be useful for this purpose? If yes, how so? If not, why?

5. For systems where you participated in preparing, reviewing, assessing, or managing the assurance case, what types of system(s) or system functions were within the scope of the case? For example, you might have been involved with a case for the whole system, a specific sub-system, component, or function of a larger system, only software, only hardware, etc.

A.3 Detailed Code Book

The following table contains the detailed codebook produced from pair coding of interview transcripts.

Category	Code	Code Definition	# Part.
Motivation for preparing ACs	Capture Arguments	Document assurance arguments in a structured or systematic manner.	12
	Compliance	Satisfy, or check compliance with, technical standards, regulatory requirements, or contractual obligations.	12
	Live Model of Assurance	Maintain a "live" model of an AC that is updated periodically, perhaps evolving alongside the system(s).	6
	Evidence Adequacy	Determine if the available evidence is adequate to support the assurance argument(s), including the possibility of making a "self-checklist" of the required evidence.	4
	Holistic Assurance	Develop an overall view of assurance, across multiple parts of an organization or a system, especially where the different parts might have different objectives (i.e., ACs help address assurance during system integration).	4
	Lack of Guidance	Lack of prescriptive guidance for novel or complex systems, so it is necessary to make an argument about why the a system meets its stated objective(s).	3
	Risk Management	Understand and manage risk associated with a system, including hazard identification, risk analysis, and mitigations.	3
	Gap Analysis	Discover or understand "gaps" in (previously implicit) assurance arguments or assurance-related activities.	≤ 2
	Complexity of Systems	Manage the increasing complexity of systems, which now require more sophisticated assurance arguments.	≤ 2
	Deployment Decision	Decide to deploy a system, based on the argument(s) and evidence presented in the AC.	≤ 2

Category	Code	Code Definition	# Part.
Expression of ACs	GSN	Goal Structuring Notation (GSN), or a GSN-like notation.	15
	Narratives	Narrative, report, or "essay" formats.	13
	CAE	Claims-Argument-Evidence (CAE) notation.	6
	EA	Eliminative Argumentation (EA) notation.	5
	Tabular	Tabular format (e.g., a spreadsheet).	≤ 2
	FAN	Friendly Argument Notation (FAN).	≤ 2
	SACM	Structured Assurance Case Meta-Model (SACM).	≤ 2
Quality ***Attributes*** for ACs	Safety	Safety is the main quality attribute being considered by the AC.	16
	Security	Security is the main quality attribute being considered by the AC.	4
	Mixing Safety & Security	A combination of safety and security is being considered by the AC.	≤ 2
	Mixing Many Attributes	A combination of many quality attributes (safety, security, reliability, availability, fitness-for-purpose, etc.) are considered by the AC.	11
Scope of ACs	Whole System	Scope of the AC is a whole system.	9
	Sub-System	Scope of the AC is a sub-system.	≤ 2
	Component(s)	Scope of the AC is a component.	≤ 2

Category	Code	Code Definition	# Part.
	Software-only	Scope of the AC is a software-only element.	≤ 2
Methods for Confidence Assessment	Review	Participant has using peer or independent review as a means of assessing or increasing confidence in an AC, including "recurring reviews" at key milestones	19
	Defeaters	Participant has experience using dialectic approaches (i.e., defeaters, doubts, challenges, etc.) to increase confidence in an AC.	12
	Checklists	Participant has experience assessing confidence in an AC by checking it against external authorities such as technical standards or guidelines ("did we think of that?").	9
	Personnel	Participant has experience assessing confidence in an AC by reasoning about the qualities of the personnel who worked on the AC.	5
	Interrogation	Participant has experience interrogating the argument structure (premises $\rightarrow$ conclusion), either using a systematic method (e.g., SPRY/iTest), guidewords, or informally as a means of self-organizing a review.	5
	Confidence Arguments	Participant has experience preparing supplementary/supporting confidence arguments focused on the confidence in the main argument (e.g., Assurance Claim Points).	3
	Counting Ratios	Participant has experience assessing confidence based on ratios derived from counting elements in the argument structures (e.g., number of defeaters v. claims).	3
	Assurance 2.0	Participant has experience applying the Assurance 2.0 method to assess and increase confidence in an AC.	≤ 2
	BNNs	Participant has experience applying Bayesian Belief Networks (BBNs) to assess confidence in an AC.	≤ 2

Category	Code	Code Definition	# Part.
	Coarse Belief	Participant has experience using coarse belief measures (e.g., “gut feel”, “on a scale from 0 - 10...”).	≤ 2
	Performance Indicators	Participant has experience using safety or key performance indicators, based on engineering or operational data, to assess confidence in an argument.	≤ 2
	DST	Participant has experience using Dempster-Shafer Theory based approaches for assessing confidence in an AC.	≤ 2
	Models	The participant has used models in a way that impacted their confidence in the case, either for preparing the case itself or for managing evidence.	6
	Formal Methods	The participant has used formal (mathematical or logical) methods to assess confidence in an AC, e.g., SAT solvers or theorem provers.	≤ 2
Barriers to Conducting Confidence Assessment	More Work	Participant identified that conducting a confidence assessment requires additional effort/work, which might impact project schedule or budgets.	13
	Inadequate Guidance	Participant identified that guidance or training on confidence assessments is missing, inadequate, or otherwise not accessible to practitioners, so practitioners don’t use the methods.	12
	Interpreting Results	Participant identified that it can be difficult to interpret the results produced by a confidence assessment method and make decisions based on them.	10
	Confidence in Confidence	Participant identified a meta-problem exists where an analyst might ask if they trust the assessed level of confidence (e.g., stopping rule for defeaters).	9
	Subjective	Participant identified that confidence assessment method(s) are too subjective, relying on expert opinions, which might vary significantly.	9

Category	Code	Code Definition	# Part.
	Immature	Participant identified that confidence assessment method(s) are too immature to be used in practice or depended upon to make decisions about critical systems.	8
	Safety Culture	Participant identified that poor safety culture is a barrier to performing confidence assessments.	7
	Lack of Adoption	Participant identified that confidence assessment methods are not applicable because assurance cases are not adopted in some industries or organizations.	6
	Terminology	Participant identified that different methods and approaches have different terminology and conceptual models, which makes them difficult to apply for practitioners.	3
	Change Impact	Participant identified that it can be difficult to (re-)assess confidence when the argument changes.	≤ 2
	Tooling	Participant identified that there is a lack of mature tooling to support confidence assessment	≤ 2
	Not Required	Participant identified that confidence assessments are not required by industry standards or practices, so they don't do them.	≤ 2
	Not Integrated	Participant identified that confidence assessment method(s) are not deeply integrated into structured notations (e.g., GSN) and so can be difficult to use.	≤ 2
	Project Management	It can be difficult to weave confidence assessments into project management/planning (e.g., too much/too early and the results are not useful, and too late and the results are overwhelming).	≤ 2

Category	Code	Code Definition	# Part.
Uses of Confidence Assessment Results	Communication with Stakeholders	The participant has experience using the results of confidence assessment to communicate with project or system stakeholders.	14
	Addressing Issues	The participant has experience using the results from confidence assessment is used to address "issues" (or defects, problems, etc.) in the system or operations or in the AC itself.	8
	Understanding Risk	The participant has experience using the results of confidence assessment to help understand residual risk (quantitatively or qualitatively) of a system.	8
	Evaluate Argument	The participant has experience using the results of confidence assessment to evaluate the argument structure, including whether conclusions are justified.	7
	Deployment Decision	The participant has experience using the results of confidence assessment to make a system deployment decision.	6
	Evidence Evaluation	The participant has experience using the result of confidence assessment to evaluate whether evidence is suitable for the argument.	6
	Avoid Confirmation Bias	The participant has experience using the results of confidence assessment help mitigate confirmation bias.	5

Appendix B
Certus Specification in Lean4

This appendix contains the full denotational semantics for the *Certus* language authored using the Lean4 theorem prover's language and meta-programming environment. An abridged version of these semantics was presented in Chapter 5. The specification and its implementation in the TypeScript programming language are also available in an open-source repository[1].

B.1 Semantic Algebras

Several semantic algebras are defined below for key concepts in *Certus*.

Unit Interval

The unit interval, $\mathcal{I}$, is the main domain over which fuzzy sets for *Certus* are defined over. To simplify proofs in Lean4, it is defined as natural numbers from $[0, 100]$.

```
def UI_UPPER : ℕ := 100
def Domain: Finset ℕ := Finset.range (UI_UPPER + 1)
abbrev UI : Type := {n : ℕ // n ∈ Domain}
def 𝓘 := UI

namespace UI
   def mk (r : ℕ) : UI := ⟨(min r UI_UPPER), by
         have hle : (min r UI_UPPER) ≤ UI_UPPER := min_le_right _ _
         have hlt : (min r UI_UPPER) < (UI_UPPER + 1) := Nat.lt_succ_of_le hle
            exact Finset.mem_range.mpr hlt⟩
   def min (a b : UI) : UI := if a.val ≤ b.val then a else b
   def max (a b : UI) : UI := if a.val ≥ b.val then a else b
   def all : Finset UI := Domain.attach
   def max_val: UI := UI.mk UI_UPPER
   instance : Fintype UI := Fintype.ofFinset Domain (by intro a rfl)
end UI
```

[1] https://gitlab.com/sdiemert/certus.

Fuzzy Sets

Operations for fuzzy sets are defined over the unit interval (`UI`).

```
def FuzzySet (a : Type u) : Type u := (x: a) → UI

namespace FuzzySet

  def intersect {α: Type} (A B : FuzzySet α): FuzzySet α :=
    λ x => UI.min (A x) (B x)
  def union {α: Type} (A B : FuzzySet α): FuzzySet α :=
    λ x => UI.max (A x) (B x)
  def cut {α: Type} [Fintype α] [DecidableEq α]
    (S : FuzzySet α) (v : UI): Finset α := by
    exact (Fintype.elems (α := α)).filter (fun x => v.val ≤ (S x).val)
  def levelMean(α : UI) (A : FuzzySet UI): ℚ :=
    let X := (UI.all).filter (λ x => (A x) ≥ α)
    let count : ℕ := X.card
    let total : ℕ := Finset.sum X (λ x => x)
    total / count
  def rank (A : FuzzySet UI): ℚ := -- Yager's fuzzy set ranking function
    (UI.all).sum (λ α => levelMean α A)
  def invert(A : FuzzySet UI): FuzzySet UI := -- Inversion (mirror image in horizontal axis)
    λ x => (A (UI.mk (UI_UPPER − x.val)))
  def complement(A : FuzzySet UI): FuzzySet UI :=
    λ x => UI.mk (UI_UPPER − (A x))

  def id {α : Type} [Fintype α] [DecidableEq α] (A B : FuzzySet α) : Bool := decide (∀ x, A x = B x)
  def eq (A B: FuzzySet UI) : Bool := decide (rank A = rank B)
  def lt (A B: FuzzySet UI) : Bool := decide (rank A < rank B)
  def gt (A B: FuzzySet UI) : Bool := decide (rank A > rank B)
  def geq (A B: FuzzySet UI) : Bool := eq A B ∨ gt A B
  def leq (A B: FuzzySet UI) : Bool := eq A B ∨ lt A B

  def subseteq {α : Type} [Fintype α] [DecidableEq α]
    (A B : FuzzySet α) : Bool := decide (∀ x, A x ≤ B x)
  def contains {α : Type} [Fintype α] [DecidableEq α]
    (A B : FuzzySet α) : Bool := decide (∀ x, A x ≥ B x)
  def some {α : Type} [Fintype α] [DecidableEq α]
    (A B : FuzzySet α) : Bool := decide (∃x, (A x).val ≠ 0 ∧ (B x).val ≠ 0)
  def isNormal {α : Type} [Fintype α] [DecidableEq α]
    (A: FuzzySet α) : Bool := decide (∃x, (A x).val >= UI_UPPER)
  def isConvex {α : Type} [Fintype α] [DecidableEq α] [LinearOrder α]
    (A: FuzzySet α) : Bool :=
    decide (∀ x y z, x ≤ y → y ≤ z → A y ≥ UI.min (A x) (A z))

  def rampUp (a b : UI) : FuzzySet UI :=
    λ x =>
    if x < a.val then <<0>>
    else if x < b.val then
      let width := b.val − a.val
      let m := UI_UPPER / width
      let y := (x − a.val) * m
      <<Nat.min UI_UPPER y>>
    else UI.max_val
  def rampDown (a b : UI) : FuzzySet UI := λ x => <<UI_UPPER − (rampUp a b) x>>
  def trap (a b c d : UI) : FuzzySet UI :=
    λ x => if x <= b then (rampUp a b) x else (rampDown c d) x
  def tri (a b c : UI) : FuzzySet UI :=
    λ x => if x <= b then (rampUp a b) x else (rampDown b c) x

end FuzzySet

abbrev 𝓕 := FuzzySet UI
```

The canonical sets are defined as follows:

```
def EmptySet : 𝓕 := λ _ => 0
```

```
def FullSet : 𝓕 := λ _ => UI_UPPER
def fz_cert : 𝓕 := λ x => if x == UI_UPPER then UI_UPPER else 0
def fz_vhigh : 𝓕 := λ x => (FuzzySet.rampUp 85 90) x
def fz_high : 𝓕 := λ x => (FuzzySet.trap 70 75 90 95) x
def fz_low : 𝓕 := λ x => (FuzzySet.trap 55 60 75 80) x
def fz_uncert : 𝓕 := λ x => (FuzzySet.trap 40 45 55 60) x
def fz_zero : 𝓕 := λ x => if x == 50 then UI_UPPER else 0
def fz_skep : 𝓕 := λ x => (FuzzySet.trap 20 25 40 45) x
def fz_opp : 𝓕 := λ x => (FuzzySet.trap 5 10 25 30) x
def fz_vopp : 𝓕 := λ x => (FuzzySet.rampDown 5 10) x
def fz_reject : 𝓕 := λ x => if x == 0 then UI_UPPER else 0
```

Arguments

The domain model for `Arguments` consist of `Nodes`, `Indicators`, and `Artifacts`. It also includes a `NodeType` and a secondary wrapper type called `Passable` that is used for passing arguments to user-defined operations.

```
abbrev NodeId := String

inductive NodeType
  | Premise
  | Defeater
  deriving DecidableEq

inductive Passable
  | node: NodeType → Passable
  | Fuzzy
  | Any
  deriving DecidableEq

structure Node where
  id : NodeId
  nodeType : NodeType
  children : List Node
  expr : Stmt
  indicators : List Indicator
  artifacts : List Artifact

structure Argument where
  root : Option Node

namespace Argument
  def init : Argument := {root := none}
  def setRoot (A : Argument) (n : Node) : Argument := { root := n }
end Argument
```

Indicators

The domain model for `Indicators` is provided below; there are no operations.

```
inductive IndicatorCategory
  | warning
  | ok
  deriving DecidableEq

structure Indicator where
  (name: String)
  (curr: ℕ)
  (valuation: ℕ → IndicatorCategory)
```

Artifacts

The domain model for `Artifacts` is provided below; there are now operations.

```
structure Artifact where
  (name: String)
```

User-Defined Operators

The domain model for user-defined operators is:

```
structure UserOp where
  (name: String)
  (args: List (String × Passable))
  (expr: FuzzyExpr)
```

The *Certus* Context (`CertusCtx`)

The domain model and operations are provided below for the `CertusCtx`.

```
structure CertusCtx where
  (map : String → Option 𝓕)
  (children: String → Option Node)
  (root: Option Node)
  (ops: String → Option UserOp)
  (indicators: String → Option Indicator)
  (artifacts: String → Option Artifact)

namespace CertusCtx

  def resultKey : String := "__result__"

  def init : CertusCtx := {
    map := λ s =>
    match s.toLower with
      | "certain" => fz_cert
      | "vhigh" => fz_vhigh
      | "high" => fz_high
      | "low" => fz_low
      | "uncert" => fz_uncert
      | "zero" => fz_zero
      | "skep" => fz_skep
      | "opp" => fz_opp
      | "vopp" => fz_vopp
      | "reject" => fz_reject
      | _ => .none
    ops := λ _ => .none
    children := λ _ => .none
    indicators := λ _ => .none
    artifacts := λ _ => .none
    root := .none }

  def get (σ : CertusCtx) (s: String) : Option 𝓕 := σ.map s
  def getOp (σ : CertusCtx) (s: String) : Option UserOp := σ.ops s
  def set (σ : CertusCtx) (s: String) (fz: 𝓕): CertusCtx := {
    map := λ s' => if s = s' then fz else σ.get s',
    ops := λ x => σ.getOp x,
    root := σ.root,
    children := σ.children,
    indicators := σ.indicators,
    artifacts := σ.artifacts }

  def setOp (σ : CertusCtx) (s: String) (op: UserOp): CertusCtx := {
    map := λ x => σ.get x,
```

```
    ops := λ s' => if s = s' then op else σ.getOp s',
    root := σ.root,
    children := σ.children,
    indicators := σ.indicators,
    artifacts := σ.artifacts }

  def addChild (σ : CertusCtx) (n: Node): CertusCtx := {
    map := λ x => σ.get x,
    ops := λ x => σ.getOp x,
    children := λ n' => if n.id = n' then n else σ.children n',
    root := σ.root,
    indicators := σ.indicators,
    artifacts := σ.artifacts }

  def setRoot (σ : CertusCtx) (r: Node): CertusCtx := {
    map := λ x => σ.get x,
    ops := λ x => σ.getOp x,
    children := σ.children,
    indicators := σ.indicators,
    artifacts := σ.artifacts,
    root := r }

  def setMeasure (σ : CertusCtx) (m: Measure): CertusCtx := {
    map := λ x => σ.get x,
    ops := λ x => σ.getOp x,
    children := σ.children,
    root := σ.root
    artifacts := σ.artifacts,
    indicators := λ m' => if m.name = m' then m else σ.indicators m' }

  def setArtifact (σ : CertusCtx) (a: Artifact): CertusCtx := {
    map := λ x => σ.get x,
    ops := λ x => σ.getOp x,
    children := σ.children,
    indicators := σ.indicators,
    root := σ.root
    artifacts := λ a' => if a.name = a' then a else σ.artifacts a' }

  def checkOpTypes (spec : Passable) (act : Passable) : Bool :=
    match spec, act with
    | .Any, _  => true
    | .Fuzzy, .Fuzzy   => true
    | .node .Premise, .node .Premise => true
    | .node .Defeater, .node .Defeater => true
    | _, _ => false

end CertusCtx
```

B.2 Abstract Syntax Models

Certus' abstract syntax models represent expressions, which are defined as Lean4 inductive types. Some of these models are `mutual` (i.e., they refer to each other) and must be wrapped in a `mutual` ... `end` block to allow Lean4 to produce proofs.

Fuzzy Expressions

Fuzzy expressions (`FuzzyExpr`) produce a fuzzy set, i.e., a value in $\mathcal{F}$. This abstract syntax model defines the operations that a user of *Certus* can perform in the language.

```
inductive FuzzyExpr
  | var (n : String)
```

```
  | invert (e : FuzzyExpr)
  | complement (e : FuzzyExpr)
  | intersect (e1 e2 : FuzzyExpr)
  | union (e1 e2 : FuzzyExpr)
  | min (E : List FuzzyExpr)
  | max (E : List FuzzyExpr)
  | trap (a b c d : UI)
  | tri (a b c : UI)
  | up (a b : UI)
  | down (a b : UI)
  | cases (E : List (BoolExpr × FuzzyExpr)) (default : FuzzyExpr)
  | invokeOp (opName: String) (params: List (String))
```

Boolean Expressions

Boolean expressions (`BoolExpr`) produce a Boolean result. This abstract syntax model defines the Boolean operators that a user can invoke *Certus*. These operators appear as part of the conditional in each case of a `cases` statement.

```
inductive BoolExpr
  | id (e1 e2 : FuzzyExpr)
  | eq (e1 e2 : FuzzyExpr)
  | gt (e1 e2 : FuzzyExpr)
  | lt (e1 e2 : FuzzyExpr)
  | geq (e1 e2 : FuzzyExpr)
  | leq (e1 e2 : FuzzyExpr)
  | is (e1 e2 : FuzzyExpr)
  | overlaps (e1 e2 : FuzzyExpr)
  | contains (e1 e2 : FuzzyExpr)
  | parens (e : BoolExpr)
  | not (e : BoolExpr)
  | and (e1 e2: BoolExpr)
  | or (e1 e2: BoolExpr)
  | indicator (e : IndicatorExpr)
  | has (a : String)
```

Indicator Expressions

Indicator expressions (`IndicatorExpr`) produce a Boolean value, based on the value of an Indicator in the current context.

```
inductive IndicatorExpr
  | is (n : String) (c: IndicatorExpr)
  | eqNum (n : String) (val: ℕ)
  | gtNum (n : String) (val: ℕ)
  | geqNum (n : String) (val: ℕ)
  | ltNum (n : String) (val: ℕ)
  | leqNum (n : String) (val: ℕ)
  | eq (n1 n2 : String)
  | gt (n1 n2 : String)
  | geq (n1 n2 : String)
  | lt (n1 n2 : String)
  | leq (n1 n2 : String)
```

Statements

Statements (`Stmt`) represent whole expressions provided by a user. Statements may be composed according to the grammar.

```
inductive Stmt
  | is (i1: String) (e2: FuzzyExpr)
  | compose (S : List Stmt)
  | userOp (op: UserOp)
```

B.3 Grammar

Grammars in Lean4 have three parts: syntax categories (defining non-terminal symbols), production rules, and then expression wrappers which are used for mapping from the production rules to the abstract syntax model.

Syntax Categories

Syntax categories correspond to non-terminal symbols:

```
declare_syntax_cat indCat
declare_syntax_cat iId
declare_syntax_cat artifactId
declare_syntax_cat indComp
declare_syntax_cat invokedArg
declare_syntax_cat opArgType
declare_syntax_cat typedParam
declare_syntax_cat fuzz
declare_syntax_cat bool
declare_syntax_cat definition
declare_syntax_cat assign
declare_syntax_cat localBind
declare_syntax_cat globalBind
declare_syntax_cat stmt
```

Production Rules

The grammar's production rules are below. Precedence is indicated with a number on each rule. Higher numbers correspond to higher precedence rules.

```
syntax:95 "warning" : indCat
syntax:95 "ok" : indCat
syntax:93 "!" noWs ident : iId
syntax:93 "@" noWs ident : artifactId
syntax:92 iId "is" indCat : indComp
syntax:92 iId "=" num : indComp
syntax:92 iId ">" num : indComp
syntax:92 iId ">=" num : indComp
syntax:92 iId "<" num : indComp
syntax:92 iId "<=" num : indComp
syntax:92 iId "=" iId : indComp
syntax:92 iId ">" iId : indComp
syntax:92 iId ">=" iId : indComp
syntax:92 iId "<" iId : indComp
syntax:92 iId "<=" iId : indComp
syntax:90 fuzz "==" fuzz : bool
syntax:90 fuzz "=" fuzz : bool
syntax:90 fuzz ">" fuzz : bool
```

```
syntax:90 fuzz ">=" fuzz : bool
syntax:90 fuzz "<" fuzz : bool
syntax:90 fuzz "<=" fuzz : bool
syntax:90 fuzz "is" fuzz : bool
syntax:90 fuzz "overlaps" fuzz : bool
syntax:90 fuzz "contains" fuzz : bool
syntax:90 "has" artifactId : bool
syntax:89 indComp : bool
syntax:80 ident : fuzz
syntax:62 "trap" "(" num "," num "," num "," num ")": fuzz
syntax:60 "tri" "(" num "," num "," num ")": fuzz
syntax:61 "up" "(" num "," num ")": fuzz
syntax:60 "down" "(" num "," num ")": fuzz
syntax:56 "invert" "(" fuzz ")": fuzz
syntax:55 "comp" "(" fuzz ")": fuzz
syntax:53 "min" "(" sepBy1(fuzz, ",") ")": fuzz
syntax:53 "max" "(" sepBy1(fuzz, ",") ")": fuzz
syntax:51 fuzz "intersect" fuzz : fuzz
syntax:50 fuzz "union" fuzz : fuzz
syntax:40 "Any" : opArgType
syntax:40 "Fuzzy" : opArgType
syntax:40 "Premise" : opArgType
syntax:40 "Defeater" : opArgType
syntax:34 "(" bool ")" : bool
syntax:33 "not" bool : bool
syntax:31 bool "and" bool : bool
syntax:30 bool "or" bool : bool
syntax:26 ident : invokedArg
syntax:25 ident "(" invokedArg,*")" : fuzz
syntax:20 "cases" (bool "->" fuzz ",")+ "otherwise" fuzz : fuzz
syntax:16 ident ":" opArgType : typedParam
syntax:13 ident "is" fuzz : assign
syntax:12 ident "as" fuzz : definition
syntax:11 ident "(" typedParam,* ")" "as" fuzz : definition
syntax:10 "with" sepBy1(definition, ";") "end" : localBind
syntax:10 "global" sepBy1(definition, ";") "end" : globalBind
syntax:4 localBind assign : stmt
syntax:3 globalBind : stmt
syntax:2 assign : stmt
```

Expression Wrappers

Expression wrappers are an additional type of production rule that are used to map the grammar's main production rules to the abstract syntax model via the `macro_rules` command shown below. These rules are not considered part of the language.

```
syntax "cat" "{" indCat "}" : term
syntax "indId" "{" iId "}" : term
syntax "artifactId" "{" artifactId "}" : term
syntax "indComp" "{" indComp "}" : term
syntax "arg" "{" invokedArg "}" : term
syntax "param" "{" typedParam "}" : term
syntax "type" "{" opArgType "}" : term
syntax "bool" "{" bool "}" : term
syntax "fuzz" "{" fuzz "}" : term
syntax "definition" "{" definition "}" : term
syntax "assign" "{" assign "}" : term
syntax "local" "{" localBind "}" : term
syntax "global" "{" globalBind "}" : term
syntax "certus" "{" stmt "}" : term
```

B.4 Mapping Grammar to the AST

Lean4 provides the macro_rules operator map grammar production rules into abstract syntax models that are used for subsequent computation.

```
macro_rules
  | `(bool{has $a:artifactId}) => `(BoolExpr.has artifactId{$a})
  | `(bool{$m:indComp}) => `(BoolExpr.measure indComp{$m})
  | `(bool{$e1:fuzz == $e2:fuzz}) => `(BoolExpr.id fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz = $e2:fuzz}) => `(BoolExpr.eq fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz > $e2:fuzz}) => `(BoolExpr.gt fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz < $e2:fuzz}) => `(BoolExpr.lt fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz >= $e2:fuzz}) => `(BoolExpr.geq fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz <= $e2:fuzz}) => `(BoolExpr.leq fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz is $e2:fuzz}) => `(BoolExpr.is fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz contains $e2:fuzz}) => `(BoolExpr.contains fuzz{$e1} fuzz{$e2})
  | `(bool{$e1:fuzz overlaps $e2:fuzz}) => `(BoolExpr.overlaps fuzz{$e1} fuzz{$e2})
  | `(bool{($e1:bool)}) => `(BoolExpr.parens bool{$e1})
  | `(bool{not $e1:bool}) => `(BoolExpr.not bool{$e1})
  | `(bool{$e1:bool and $e2:bool}) => `(BoolExpr.and bool{$e1} bool{$e2})
  | `(bool{$e1:bool or $e2:bool}) => `(BoolExpr.or bool{$e1} bool{$e2})

  | `(cat{warning}) => `(IndicatorCategory.warning)
  | `(cat{ok}) => `(IndicatorCategory.ok)
  | `(indId{!$i:ident}) => `($(quote i.getId.toString))
  | `(artifactId{@$i:ident}) => `($(quote i.getId.toString))
  | `(indComp{$i:iId is $c:indCat}) => `(IndicatorExpr.is indId{$i} cat{$c})
  | `(indComp{$i:iId = $n:num}) => `(IndicatorExpr.eqNum indId{$i} $n)
  | `(indComp{$i:iId > $n:num}) => `(IndicatorExpr.gtNum indId{$i} $n)
  | `(indComp{$i:iId >= $n:num}) => `(IndicatorExpr.geqNum indId{$i} $n)
  | `(indComp{$i:iId < $n:num}) => `(IndicatorExpr.ltNum indId{$i} $n)
  | `(indComp{$i:iId <= $n:num}) => `(IndicatorExpr.leqNum indId{$i} $n)
  | `(indComp{$i:iId = $j:iId}) => `(IndicatorExpr.eq indId{$i} measureId{$j})
  | `(indComp{$i:iId > $j:iId}) => `(IndicatorExpr.gt indId{$i} measureId{$j})
  | `(indComp{$i:iId >= $j:iId}) => `(IndicatorExpr.geq indId{$i} measureId{$j})
  | `(indComp{$i:iId < $j:iId}) => `(IndicatorExpr.lt indId{$i} measureId{$j})
  | `(indComp{$i:iId <= $j:iId}) => `(IndicatorExpr.leq indId{$i} measureId{$j})

  | `(type{Premise}) => `(Passable.node .Premise)
  | `(type{Defeater}) => `(Passable.node .Defeater)
  | `(type{Fuzzy}) => `(Passable.Fuzzy)
  | `(type{Any}) => `(Passable.Any)

  | `(arg{$a:ident}) => `($(quote a.getId.toString))
  | `(fuzz{$i:ident}) => `(FuzzyExpr.var $(quote i.getId.toString))
  | `(fuzz{$e1:fuzz intersect $e2:fuzz}) => `(FuzzyExpr.intersect fuzz{$e1} fuzz{$e2})
  | `(fuzz{trap($a:num,$b:num,$c:num,$d:num)}) => `(FuzzyExpr.trap $a $b $c $d)
  | `(fuzz{tri($a:num,$b:num,$c:num)}) => `(FuzzyExpr.tri $a $b $c)
  | `(fuzz{up($a:num,$b:num)}) => `(FuzzyExpr.up $a $b)
  | `(fuzz{down($a:num,$b:num)}) => `(FuzzyExpr.down $a $b)
  | `(fuzz{invert($f1:fuzz)}) => `(FuzzyExpr.invert fuzz{$f1})
  | `(fuzz{comp($f1:fuzz)}) => `(FuzzyExpr.complement fuzz{$f1})
  | `(fuzz{min($[$f:fuzz],*)}) => `(FuzzyExpr.min [ $[fuzz{$f}],* ])
  | `(fuzz{max($[$f:fuzz],*)}) => `(FuzzyExpr.max [ $[fuzz{$f}],* ])
  | `(fuzz{$e1:fuzz union $e2:fuzz}) => `(FuzzyExpr.union fuzz{$e1} fuzz{$e2})
  | `(fuzz{cases $[$c:bool -> $r:fuzz,]* otherwise $d:fuzz}) =>
      `(FuzzyExpr.cases [ $[(bool{$c}, fuzz{$r})],* ] (fuzz{$d}))
  | `(fuzz{$op:ident($ps:invokedArg,*)})
      => `(FuzzyExpr.invokeOp $(quote op.getId.toString) [ $[arg{$ps}],* ])
  | `(param{$i:ident : $t:opArgType}) => `( ($(quote i.getId.toString), type{$t}) )
```

```
| `(definition{$i:ident ($ps:typedParam,*) as $f:fuzz} )
    => `(Stmt.userOp {
       name := $(quote i.getId.toString),
       args := [$[param{$ps}],*],
       expr := fuzz{$f}
       })
| `(definition{$i:ident as $f:fuzz}) => `(Stmt.is $(quote i.getId.toString) fuzz{$f})
| `(local{with $[$dd:definition];* end}) => `([ $[definition{$dd}],* ])
| `(global{global $[$dd:definition];* end}) => `([ $[definition{$dd}],* ])
| `(assign{$i:ident is $e:fuzz}) => `(Stmt.is $(quote i.getId.toString) fuzz{$e})
| `(certus{ $a:assign }) => `(Stmt.compose [assign{$a}])
| `(certus{ $g:globalBind }) => `(Stmt.compose global{$g})
| `(certus{ $b:localBind $a:assign }) => `(Stmt.compose (local{$b} ++ [assign{$a}]))
```

B.5 Valuation Functions

Given a `CertusCtx`, the valuation functions operate over the abstract syntax model to produce an updated `CertusCtx` that contains the results of evaluating the *Certus* expression. The valuation functions use Lean4's `.some` and `.except` operators to propagate errors.

Evaluation of an Argument

Given an `Argument` and `CertusCtx`, the `evaluate` function recursively computes the belief valuation for each step in the argument.

```
partial def evalNode (r : Node) (σ : CertusCtx) : Except String CertusCtx :=
  let σ' := CertusCtx.setRoot σ r
    |> (r.children.foldl (λ α c => CertusCtx.addChild α c) · )
    |> (r.indicators.foldl (λ α m => CertusCtx.setIndicator α m) · )
    |> (r.artifacts.foldl (λ α r => CertusCtx.setArtifact α r) · )
    |> (r.children.foldlM (λ α c => do
        let childCtx ← (evalNode c σ)
        let childVal ← match (CertusCtx.get childCtx c.id) with
                  | .some x => .ok x
                  | .none => .error "Valuation error!"
        pure (CertusCtx.set α c.id childVal)) · )
  match σ' with
    | .ok α => (certusEval α r.expr)
    | .error e => .error e

def evaluate (A : Argument) (σ : CertusCtx) : Except String CertusCtx :=
  match A.root with
    | .some r => (evalNode r σ)
    | .none => .error "No root provided!"
```

Evaluation of a *Certus* Expression for One Argument Step

The function `certusEval` evaluates a *Certus* expression for one step in the argument. It has two helper evaluation functions for handling `is` statements and user-defined operators.

```
def certusEval (σ : CertusCtx) : Stmt → Except String CertusCtx
  | .is i1 e2 => evalStmtIs σ i1 e2
  | .compose S => S.foldl (λ α s =>
                      match α with
                        | .ok acc => (certusEval acc s)
                        | .error e => .error e)
                      (.ok σ)
  | .userOp x => evalUserOp σ x

def evalStmtIs (σ : CertusCtx) : String → FuzzyExpr → Except String CertusCtx :=
  λ i e =>
    let x := (evalFuzzy σ MAX_CALL_STACK_DEPTH e)
    match x with
      | .ok ret =>
        let σ' := σ.set i ret
        let σ'' := σ'.set CertusCtx.resultKey ret
          .ok σ''
      | .error e => .error e

def evalUserOp (σ : CertusCtx): UserOp → Except String CertusCtx := λ op => .ok (σ.setOp op.name op)
```

Evaluation of Fuzzy Expressions

Fuzzy expressions are evaluated by the `evalFuzzy` function which matches different cases in the abstract syntax model and then recursively evaluates each operand. An additional `checkFuzzy` helper function is used to confirm that the fuzzy expression produces a normalized convex fuzzy set.

```
def checkFuzzy (expr : FuzzyExpr) (fz: Except String 𝓕) : Except String 𝓕 :=
  match fz with
    | .ok f =>
      if !f.isNormal then .error ("Expression produces sub-normal fuzzy set")
      else if !f.isConvex then .error ("Expression produces non-convex fuzzy set")
      else .ok f
    | .error e => .error e

def evalFuzzy (σ : CertusCtx) (depth : ℕ) (fz: FuzzyExpr) : Except String 𝓕 :=
  let ret := match fz with
    | .var v => match (σ.get v) with
      | .some x => .ok x
      | .none => .error ("Symbol not found in context")
    | .intersect f1 f2 => do
      let l ← evalFuzzy σ depth f1
      let r ← evalFuzzy σ depth f2
      pure (FuzzySet.intersect l r)
    | .union f1 f2 => do
      let l ← evalFuzzy σ depth f1
      let r ← evalFuzzy σ depth f2
      pure (FuzzySet.union l r)
    | .invert f1 => do
      let x ← evalFuzzy σ depth f1
      pure (FuzzySet.invert x)
    | .complement f1 => do
      let x ← evalFuzzy σ depth f1
      pure (FuzzySet.complement x)
    | .min F => match F with
      | [ ] => .error "No parameters passed to min"
      | [f] => (evalFuzzy σ depth f)
      | f :: fs =>
        let rest := (evalFuzzy σ depth (FuzzyExpr.min fs))
        let curr := (evalFuzzy σ depth f)
        match rest, curr with
          | .ok x, .ok r => if (FuzzySet.lt x r) then .ok x else .ok r
```

```
            | .error e, _ => .error e
            | _, .error e => .error e
    | .max F => match F with
      | [ ] => .error "No parameters passed to max"
      | [f] => (evalFuzzy σ depth f)
      | f :: fs =>
        let rest := (evalFuzzy σ depth (FuzzyExpr.min fs))
        let curr := (evalFuzzy σ depth f)
        match rest, curr with
          | .ok x, .ok r => if (FuzzySet.gt x r) then .ok x else .ok r
          | .error e, _ => .error e
          | _, .error e => .error e
    | .trap a b c d => if a ≤ b ∧ b ≤ c ∧ c ≤ d then .ok (FuzzySet.trap a b c d) else .error "Invalid"
    | .tri a b c => if a ≤ b ∧ b ≤ c then .ok (FuzzySet.tri a b c) else .error "Invalid"
    | .up a b => if a ≤ b then .ok (FuzzySet.rampUp a b) else .error "Invalid"
    | .down a b => if a ≤ b then .ok (FuzzySet.rampDown a b) else .error "Invalid"
    | .cases E d =>
      match E with
        | [ ] => (evalFuzzy σ depth d)
        | (e,r) :: ES => match evalBool σ depth e with
                         | .ok true => (evalFuzzy σ depth r)
                         | .ok false => (evalFuzzy σ depth (.cases ES d))
                         | .error err => .error err
    | .invokeOp opName actuals =>
      let maybeOp := σ.getOp opName
      match maybeOp with
      | .none => .error ("Operation not found in context")
      | .some op =>
        let spec : List String := op.args.map ·.(fst)
        if spec.length ≠ actuals.length
        then .error ("Incorrect number of parameters")
        else
        let fTypes : List Passable := op.args.map ·.(snd)
        let aTypes : List Passable := actuals.map (λ a
          => match (σ.children a) with | some c => .node c.nodeType | none => .Fuzzy)
        let typesOK : Bool := ((fTypes.zip aTypes).map (λ p
          => CertusCtx.checkOpTypes p.snd p.fst)).all id
        let binds : List (String × String) := spec.zip actuals
        let overlay : String → Option ℱ := λ s =>
          match binds.find? (λ p => p.fst = s) with
            | some (_, a) => (σ.get a) -- s is a parameter, get from context
            | none => -- s is not a parameter, check map for non-node symbol
              match (σ.children s) with
                | .some _ => .none -- s corresponds to a node, cannot access in user op
                | .none => (σ.get s) -- s is not a node, get value from context
        let opCtx : CertusCtx := {
          map := overlay,
          ops := σ.ops,
          root := none,
          children := λ _ => none,
          measures := σ.measures
          artifacts := σ.artifacts
        }
        if typesOK ∧ depth > 0
          then (evalFuzzy opCtx (depth − 1) op.expr)
          else .error ("Parameter type mismatch for " ++ opName)

-- Check fuzzy set validity before returning
(checkFuzzy fz ret)
```

Evaluating Boolean Expressions

Boolean expressions appear in `cases` expressions in *Certus*. The abstract syntax model is evaluated by the `evalBool` function shown below.

```
def evalBool (σ : CertusCtx) (depth : ℕ) : BoolExpr → Except String Bool
  | .id f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.id l r)
  | .eq f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.eq l r)
  | .gt f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.gt l r)
  | .lt f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.lt l r)
  | .geq f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.geq l r)
  | .leq f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.leq l r)
  | .is f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.contains r l)
  | .contains f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.contains l r)
  | .overlaps f1 f2 => do
    let l ← evalFuzzy σ depth f1
    let r ← evalFuzzy σ depth f2
    pure (FuzzySet.some l r)
  | .parens e => do
    let x ← evalBool σ depth e
    pure (x)
  | .not e => do
    let x ← evalBool σ depth e
    pure (!x)
  | .and e1 e2 => do
    let l ← evalBool σ depth e1
    let r ← evalBool σ depth e2
    pure (l && r)
  | .or e1 e2 => do
    let l ← evalBool σ depth e1
    let r ← evalBool σ depth e2
    pure (l || r)
  | .measure e => do
    let x ← (evalMeasure σ e)
    pure (x)
  | .has a =>
    match (σ.artifacts a) with
      | .some _ => .ok true
      | .none => .ok false
```

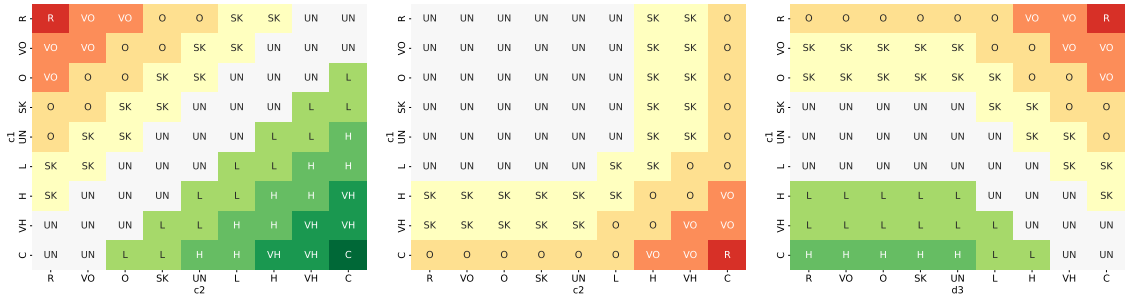
Evaluating Expressions Containing Indicators

Boolean expressions that contain indicators are evaluated by an additional function called `evalIndicator`.

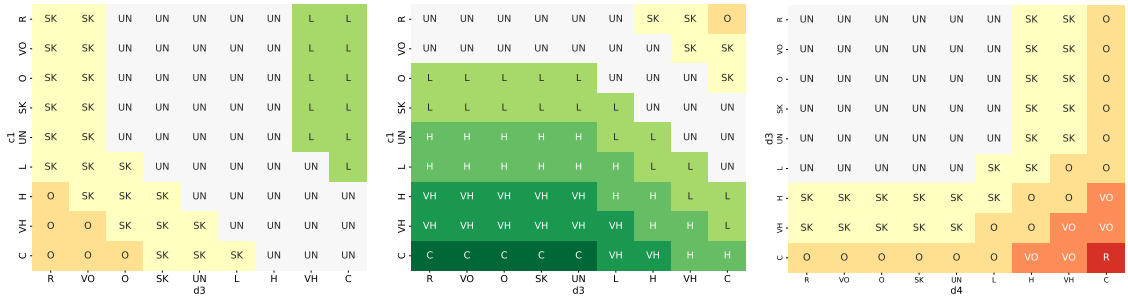
```
def getIndicator (σ : CertusCtx) (n : String) : Except String Indicator :=
  match σ.measures n with
  | .some m => .ok m
  | .none => .error ("Indicator not found in context: '" ++ n ++ "'")

def evalIndicator (σ : CertusCtx) : IndicatorExpr → Except String Bool
  | .is n c => do
    let m ← (getIndicator σ n)
    let c' ← .ok (m.valuation m.curr)
    pure (c' = c)
  | .eqNum n v => do
    let m ← getIndicator σ n
    pure (m.curr = v)
  | .gtNum n v => do
    let m ← getIndicator σ n
    pure (m.curr > v)
  | .geqNum n v => do
    let m ← getIndicator σ n
    pure (m.curr ≥ v)
  | .ltNum n v => do
    let m ← getIndicator σ n
    pure (m.curr < v)
  | .leqNum n v => do
    let m ← getIndicator σ n
    pure (m.curr ≤ v)
  | .eq n1 n2 => do
    let m1 ← getIndicator σ n1
    let m2 ← getIndicator σ n2
    pure (m1.curr = m2.curr)
  | .gt n1 n2 => do
    let m1 ← getIndicator σ n1
    let m2 ← getIndicator σ n2
    pure (m1.curr > m2.curr)
  | .geq n1 n2 => do
    let m1 ← getIndicator σ n1
    let m2 ← getIndicator σ n2
    pure (m1.curr ≥ m2.curr)
  | .lt n1 n2 => do
    let m1 ← getIndicator σ n1
    let m2 ← getIndicator σ n2
    pure (m1.curr < m2.curr)
  | .leq n1 n2 => do
    let m1 ← getIndicator σ n1
    let m2 ← getIndicator σ n2
    pure (m1.curr ≤ m2.curr)
```

Appendix C
Sensitivity Analysis Results

This appendix contains the detailed results of the sensitivity analysis for *Certus*'s macros from Section 7.3. The results are organized by dialectic interpretation and then by macro.

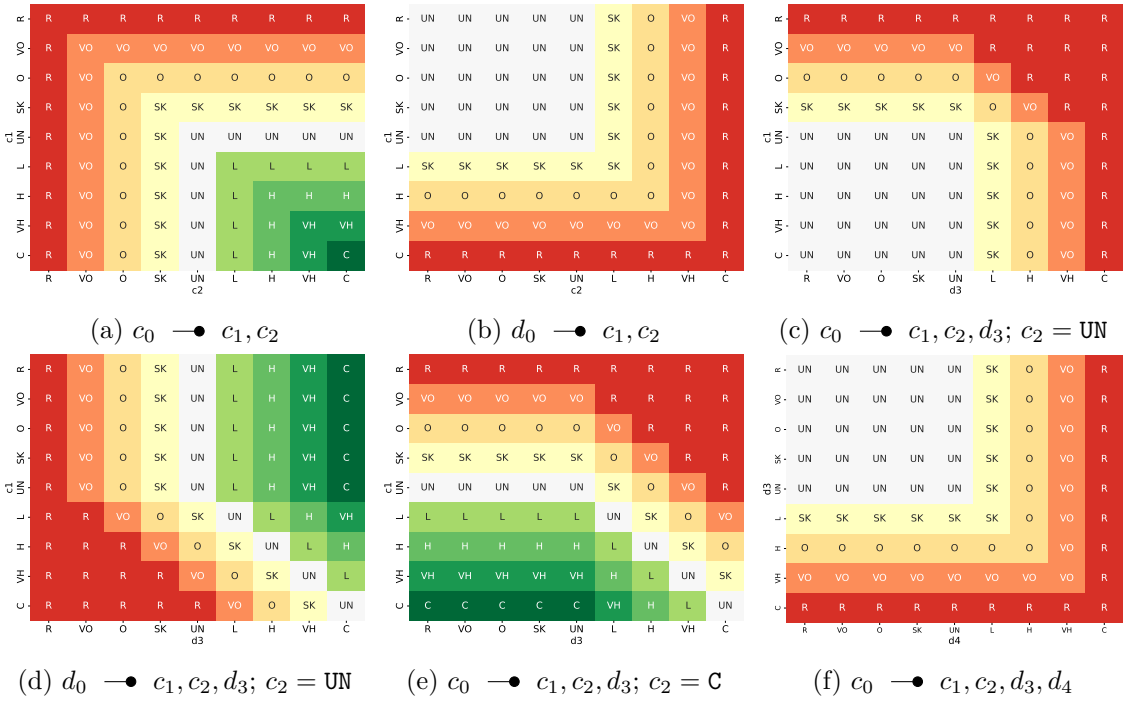


(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$



(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

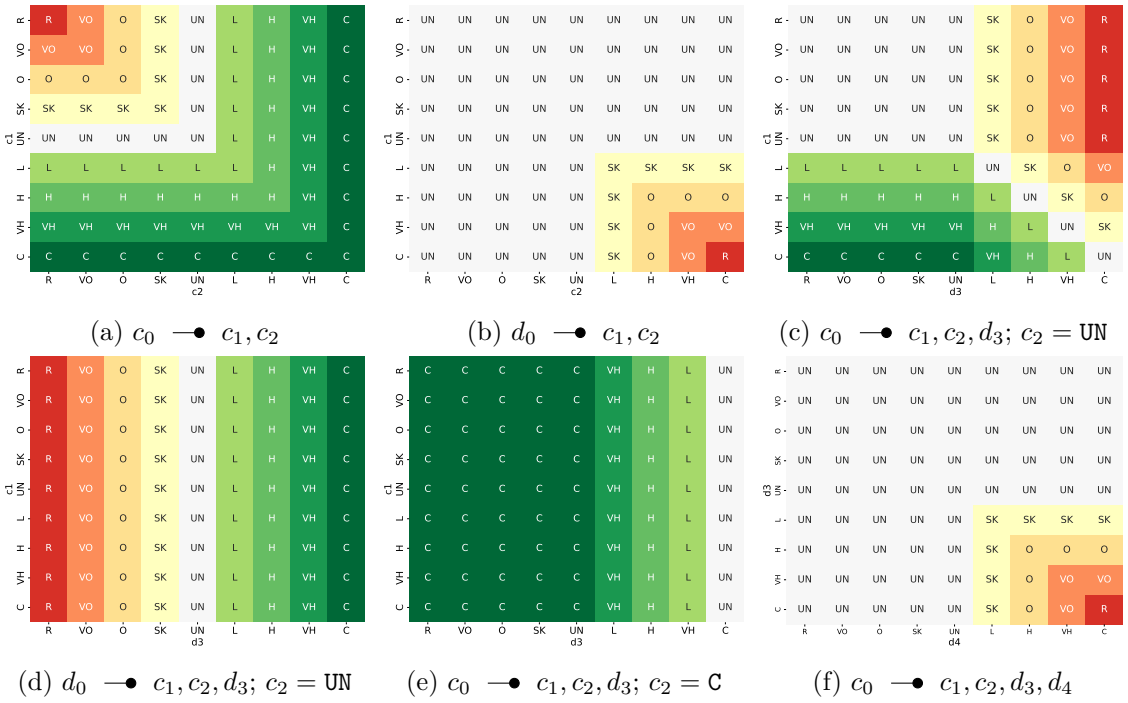
Figure C.1: Sensitivity results for the deductive #FUSE macro.



(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$

(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

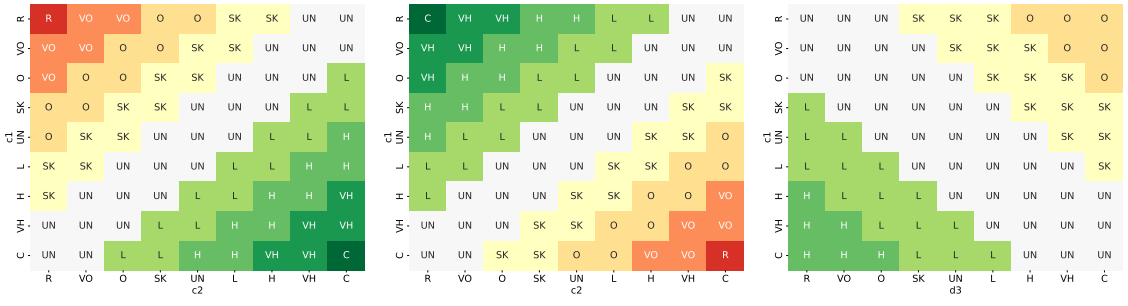
Figure C.2: Sensitivity results for the deductive #MIN macro.



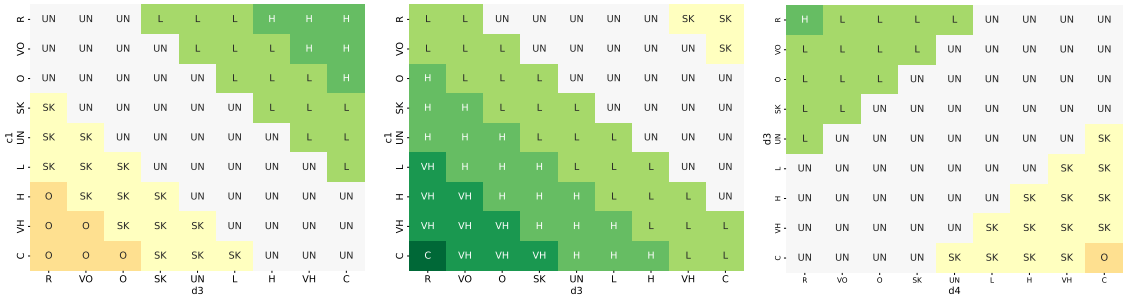
(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$

(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

Figure C.3: Sensitivity results for the deductive #MAX macro.

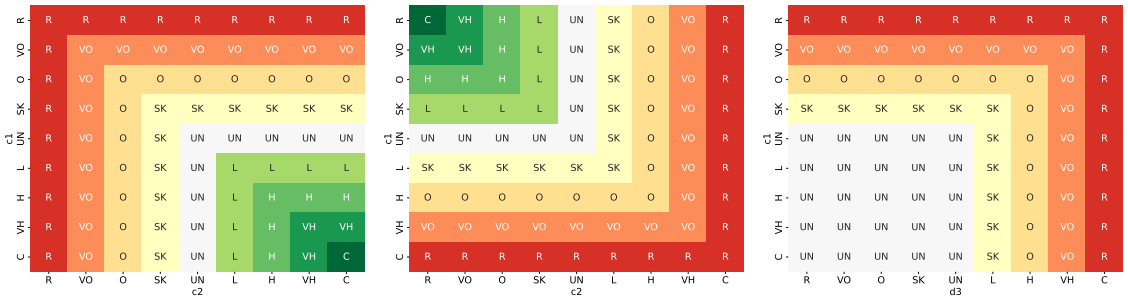


(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$

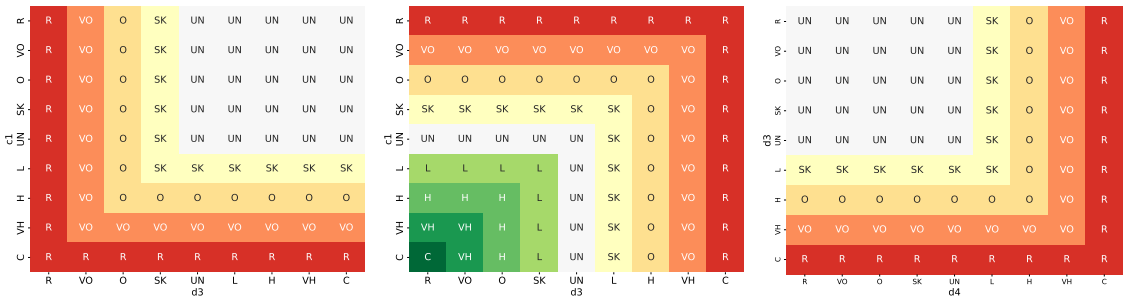


(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

Figure C.4: Sensitivity results for the eliminative #FUSE macro.

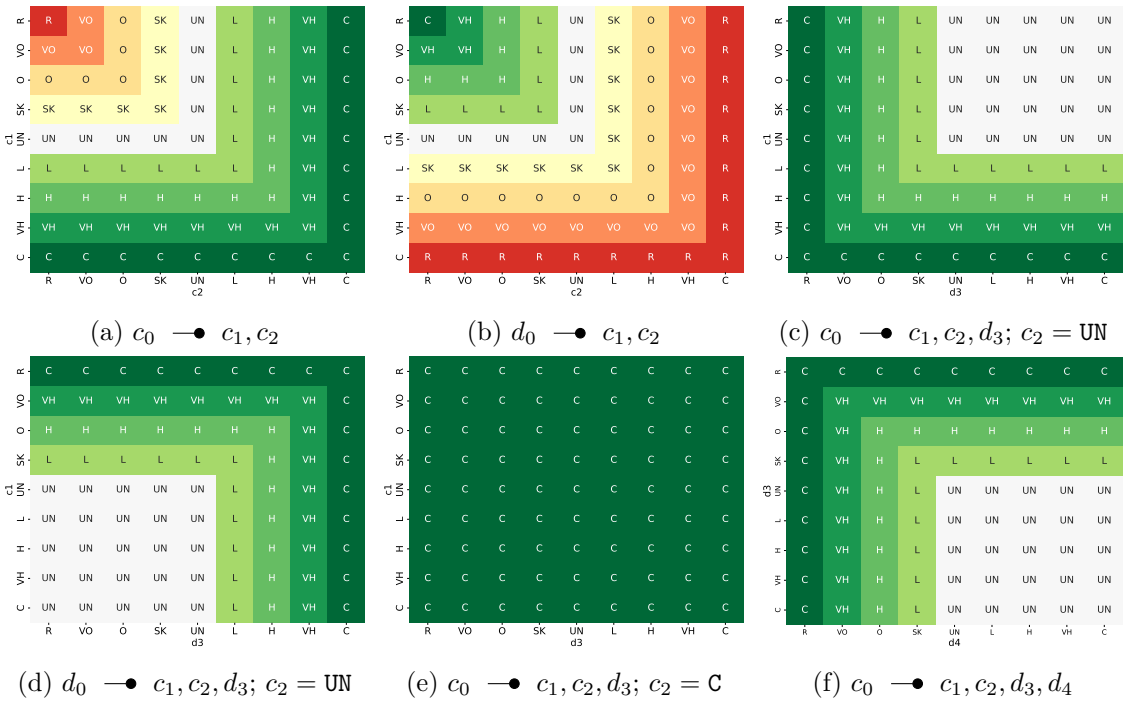


(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$



(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

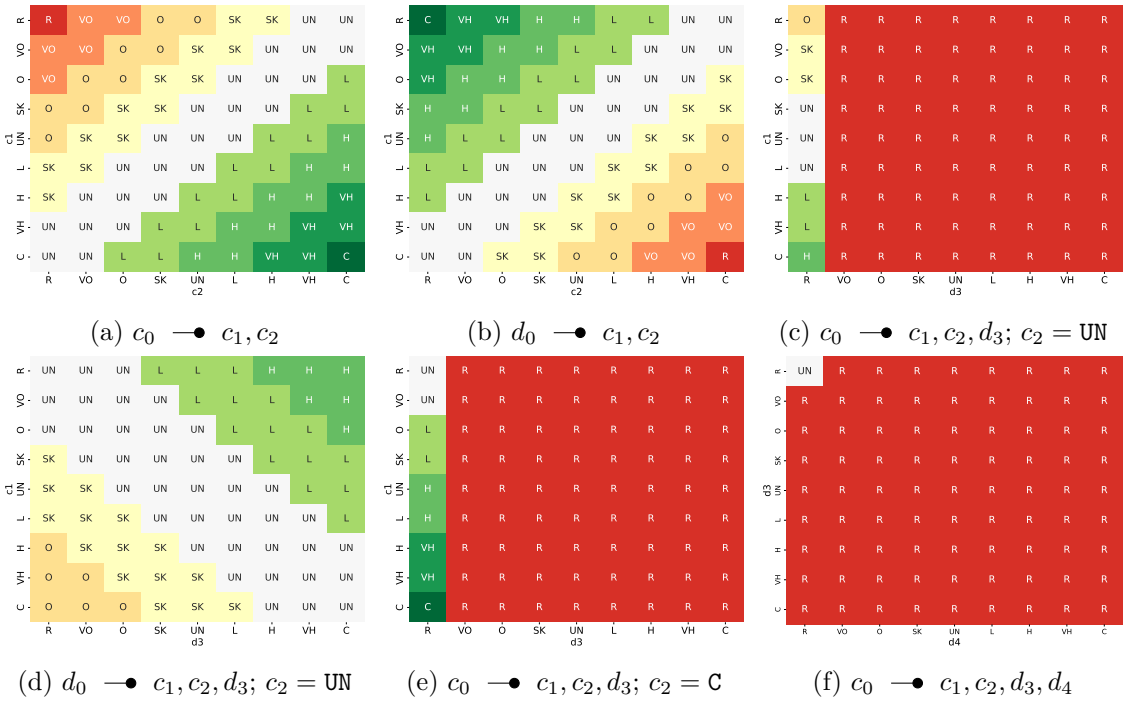
Figure C.5: Sensitivity results for the eliminative #MIN macro.



(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$

(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

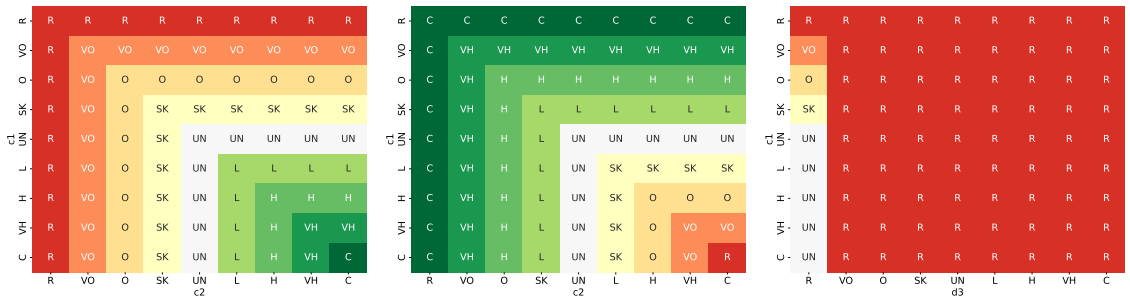
Figure C.6: Sensitivity results for the eliminative #MAX macro.



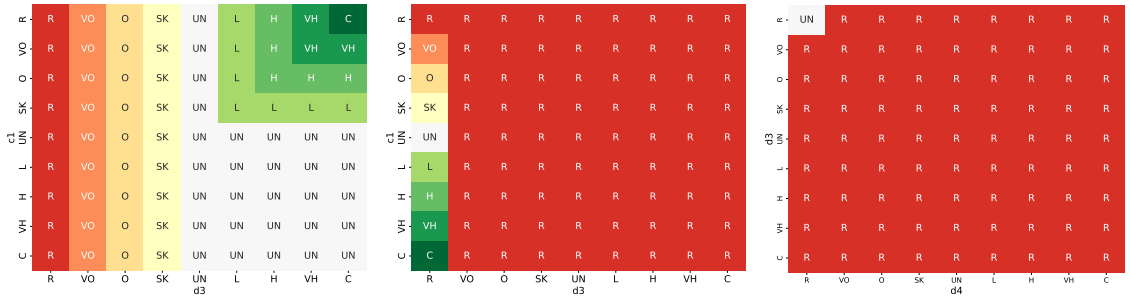
(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$

(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

Figure C.7: Sensitivity results for the strict #FUSE macro.

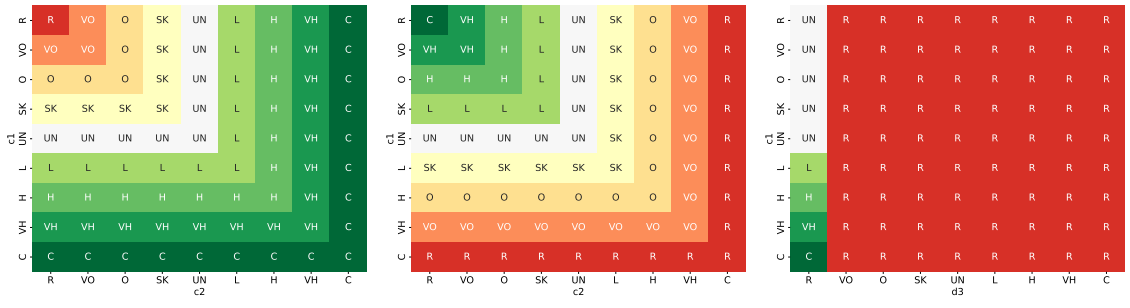


(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$

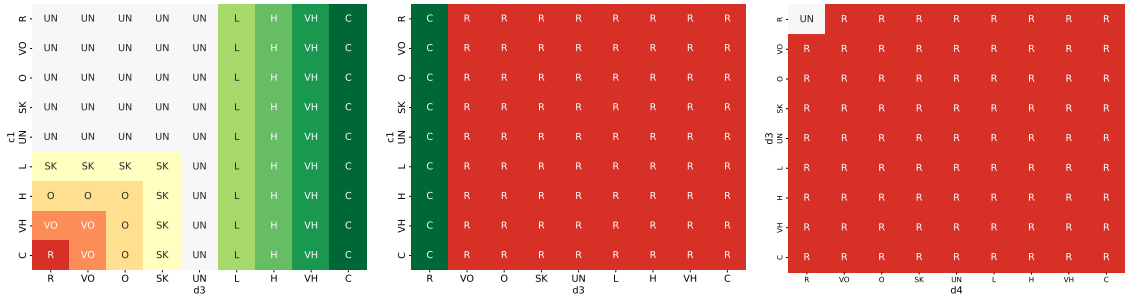


(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

Figure C.8: Sensitivity results for the Strict #MIN macro.



(a) $c_0 \multimap c_1, c_2$ (b) $d_0 \multimap c_1, c_2$ (c) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$



(d) $d_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{UN}$ (e) $c_0 \multimap c_1, c_2, d_3;\ c_2 = \texttt{C}$ (f) $c_0 \multimap c_1, c_2, d_3, d_4$

Figure C.9: Sensitivity results for the strict #MAX macro.

Appendix D
Expressivity Analysis Results

This appendix contains the detailed results from the expressivity analysis for *Certus*'s macros from Section 7.4. The results are shown as argument fragments expressed in the EA notation.

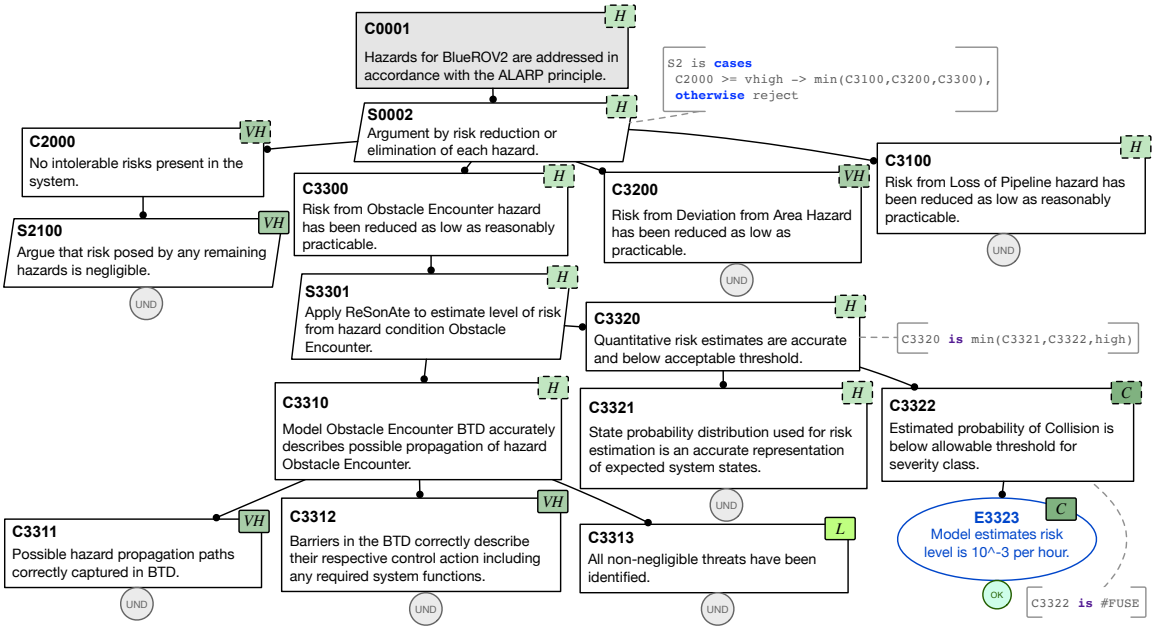


Figure D.1: Expressivity analysis result for the BLUEROV argument fragment [212].

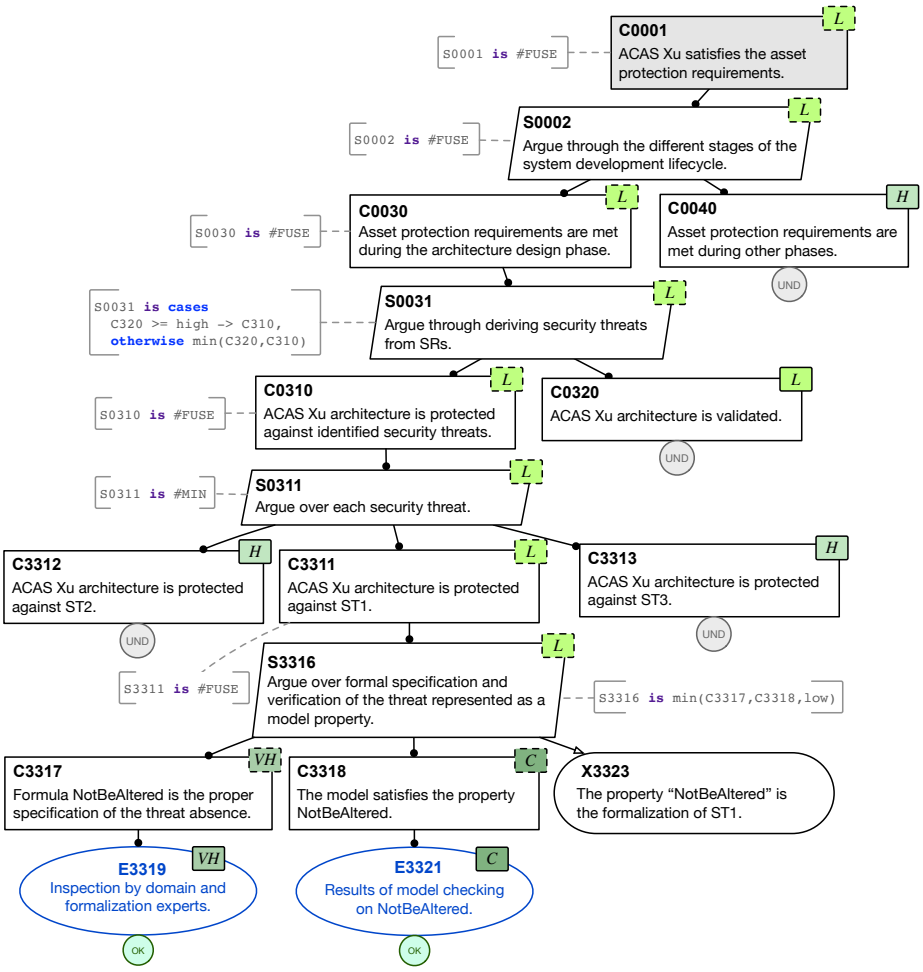


Figure D.2: Expressivity analysis result for the ACAS-Xu argument fragment [211].

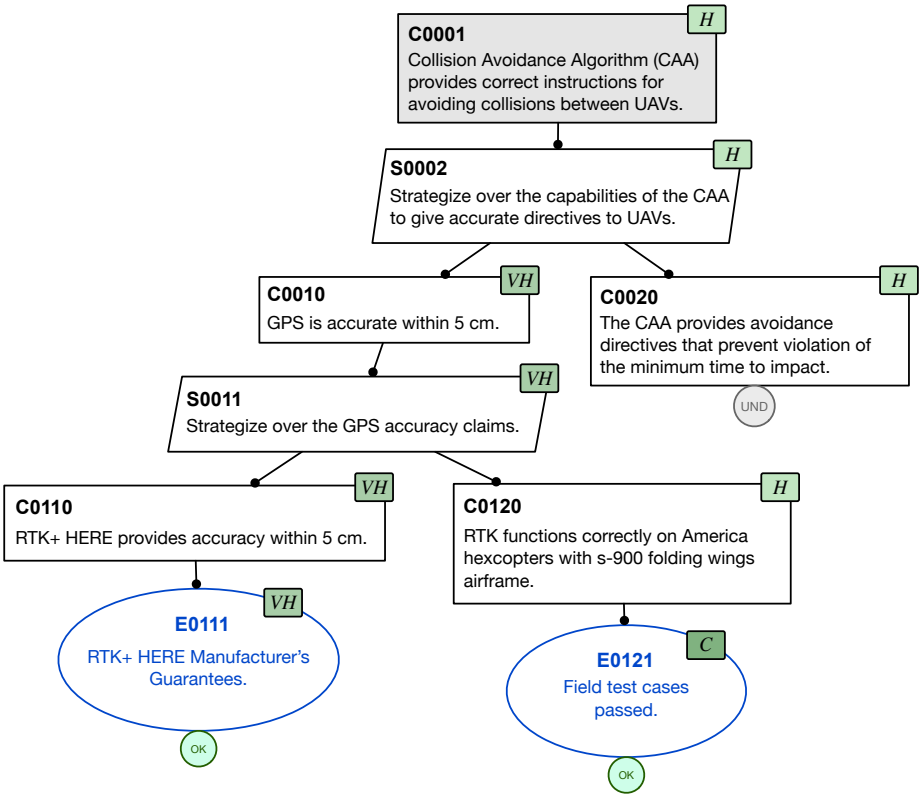


Figure D.3: Expressivity analysis result for the CAA-1 argument fragment [213].

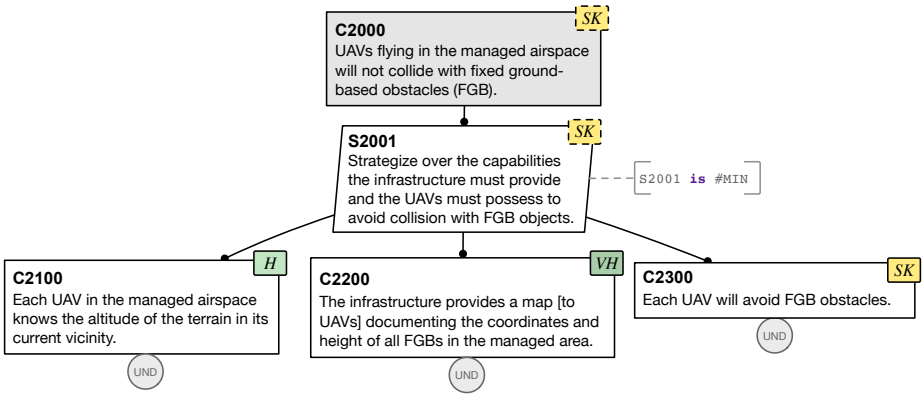


Figure D.4: Expressivity analysis result for the CAA-2 argument fragment [213].

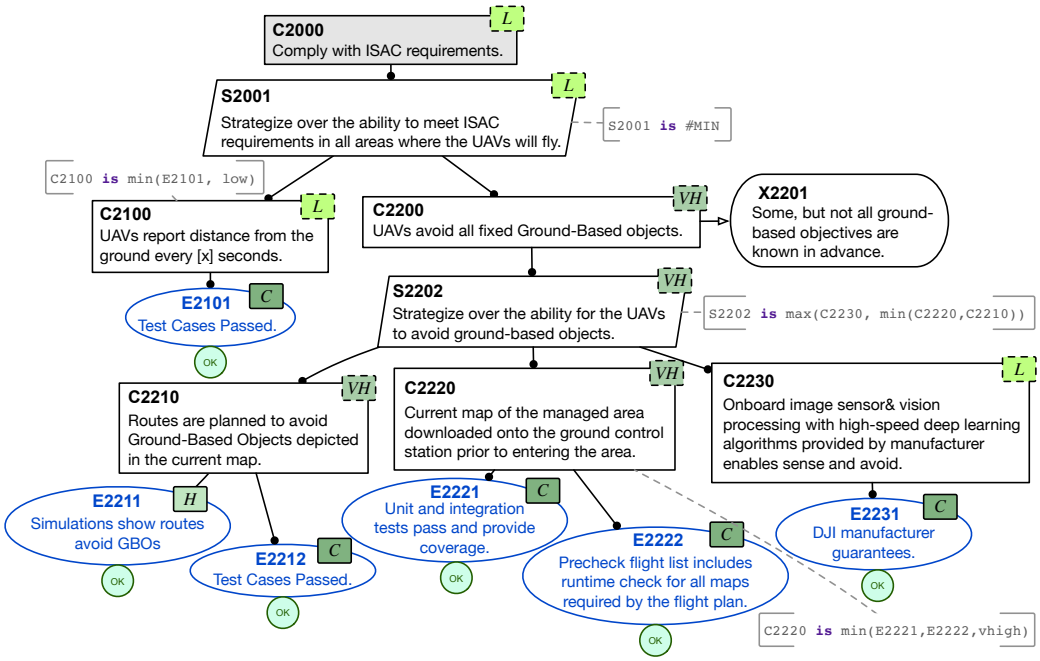


Figure D.5: Expressivity analysis result for the CAA-3 argument fragment [213].

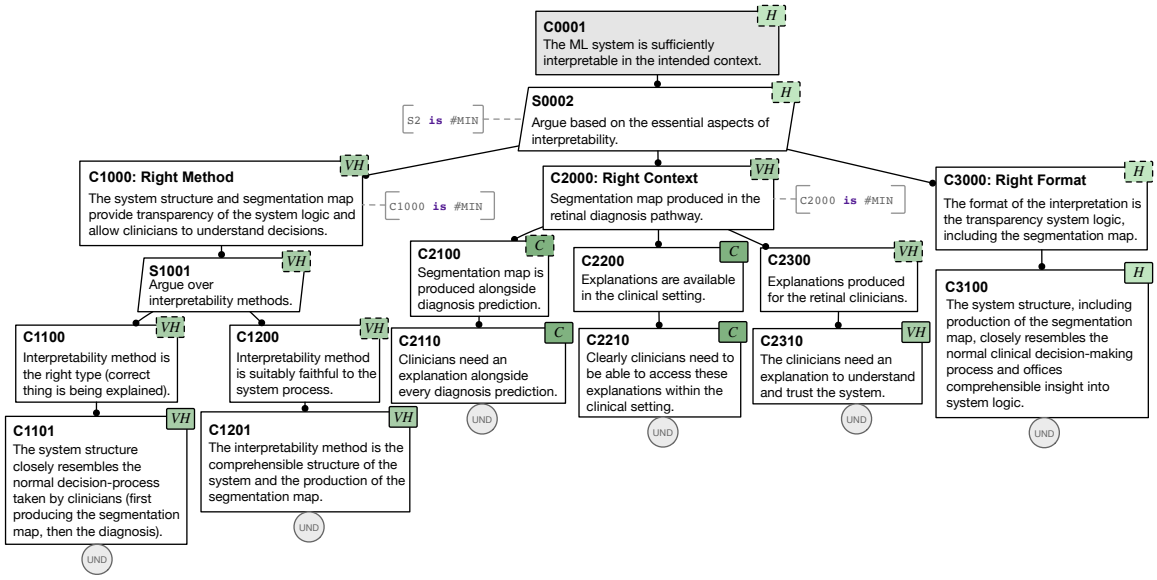


Figure D.6: Expressivity analysis result for the DeepMind argument fragment [214].

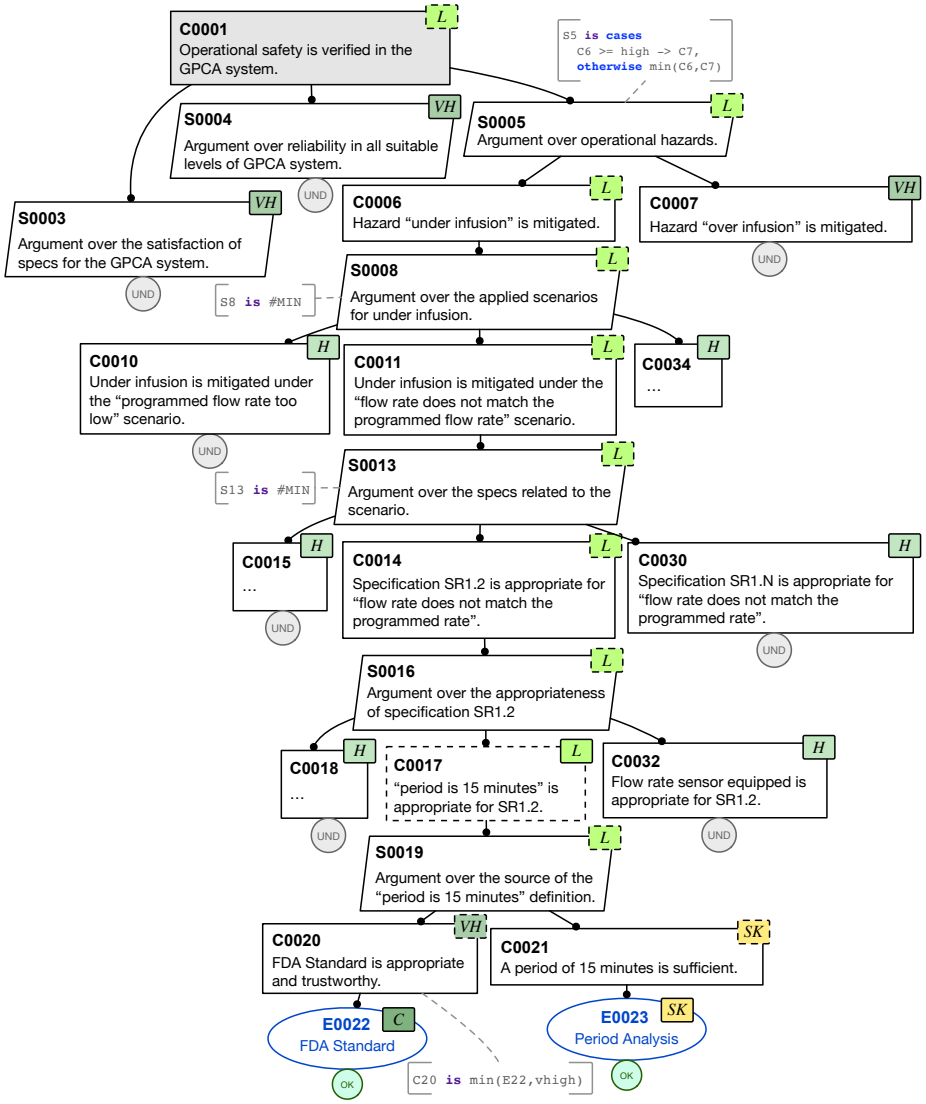


Figure D.7: Expressivity analysis result for the GPCA argument fragment [215].

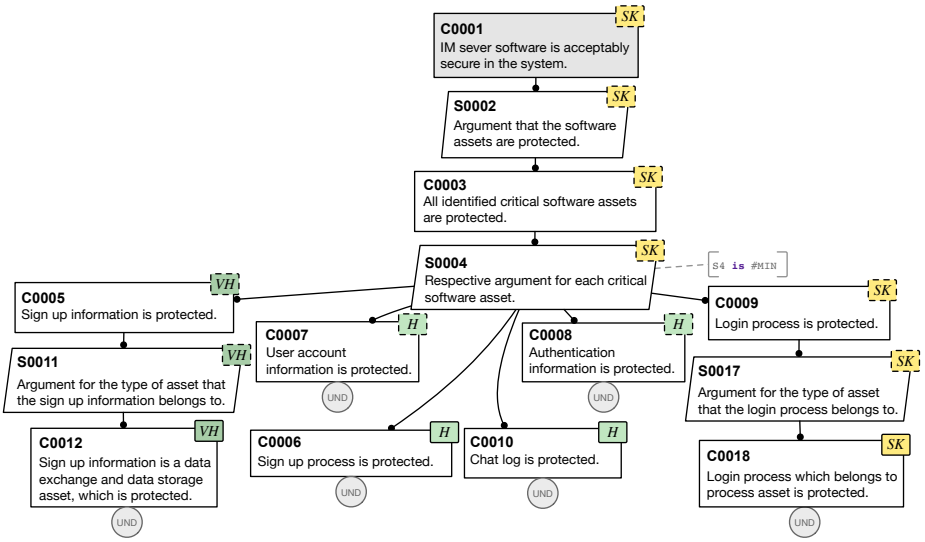


Figure D.8: Expressivity analysis result for the IM-Server argument fragment [216].

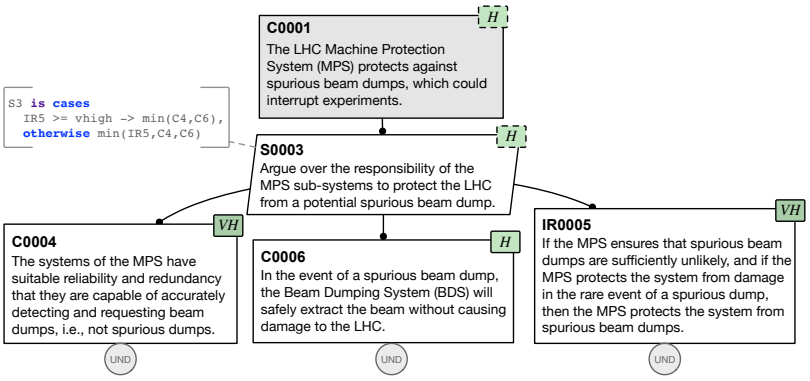


Figure D.9: Expressivity analysis result for the CERN-2 argument fragment [91].

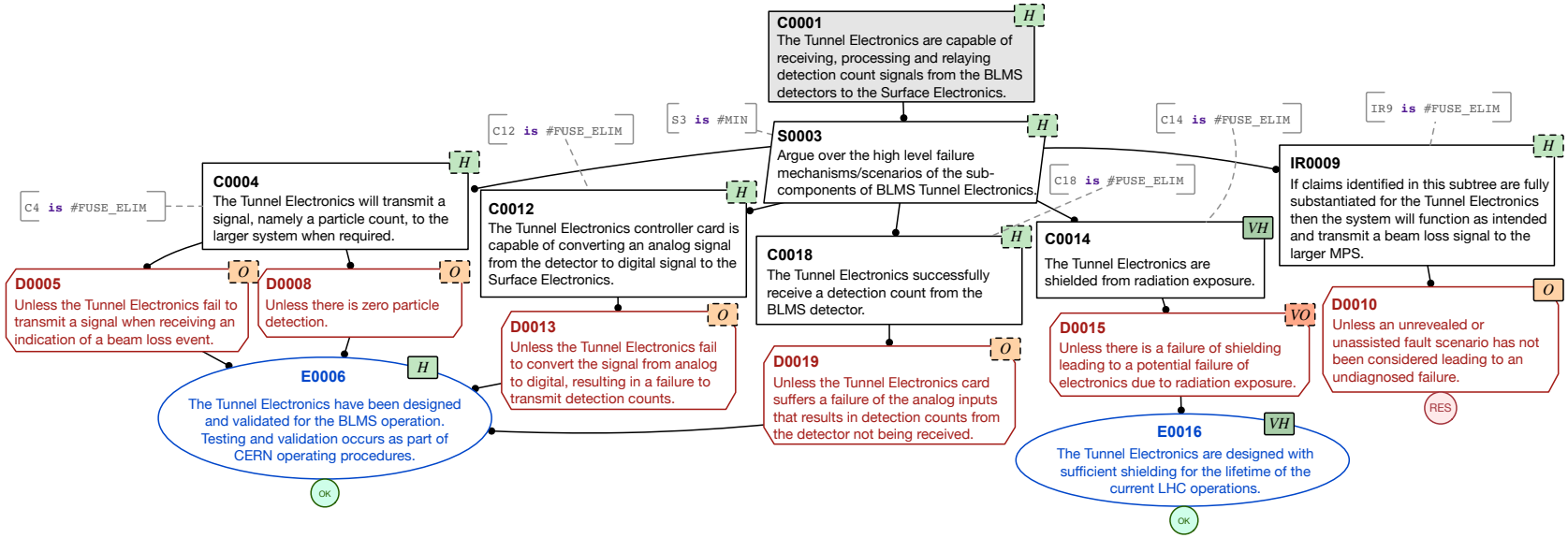


Figure D.10: Expressivity analysis result for the CERN-1 argument fragment [91].

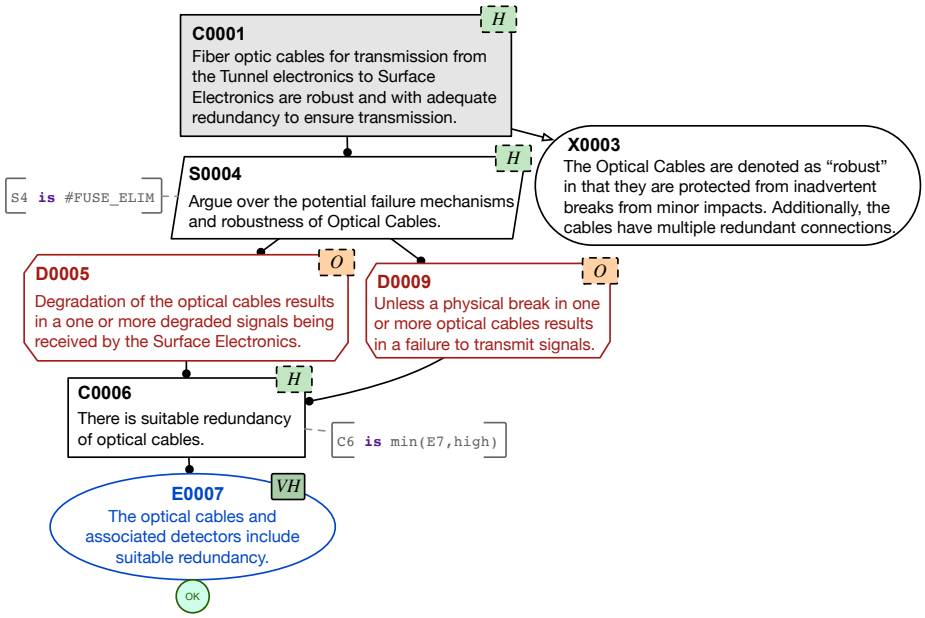


Figure D.11: Expressivity analysis result for the CERN-3 argument fragment [91].

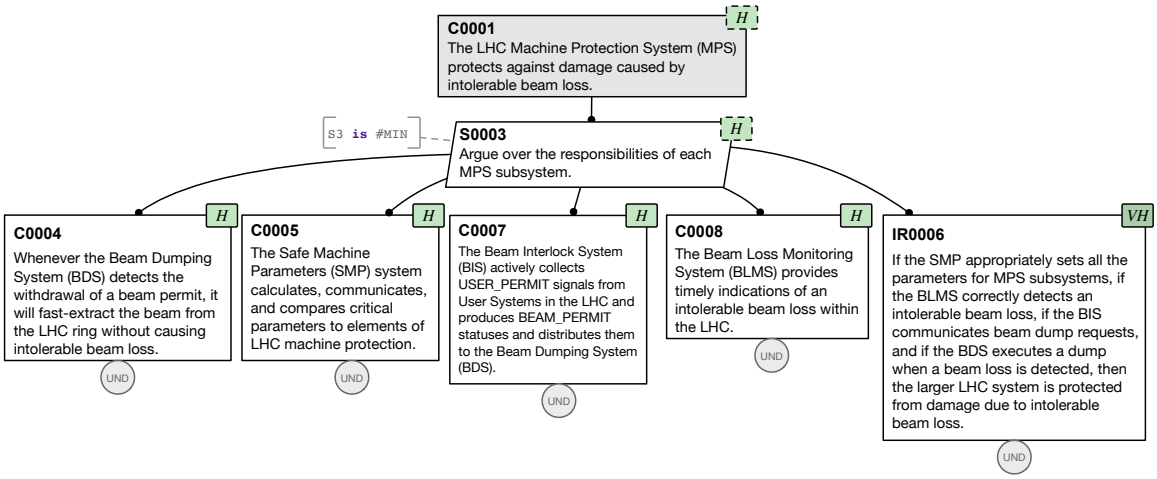


Figure D.12: Expressivity analysis result for the CERN-4 argument fragment [91].

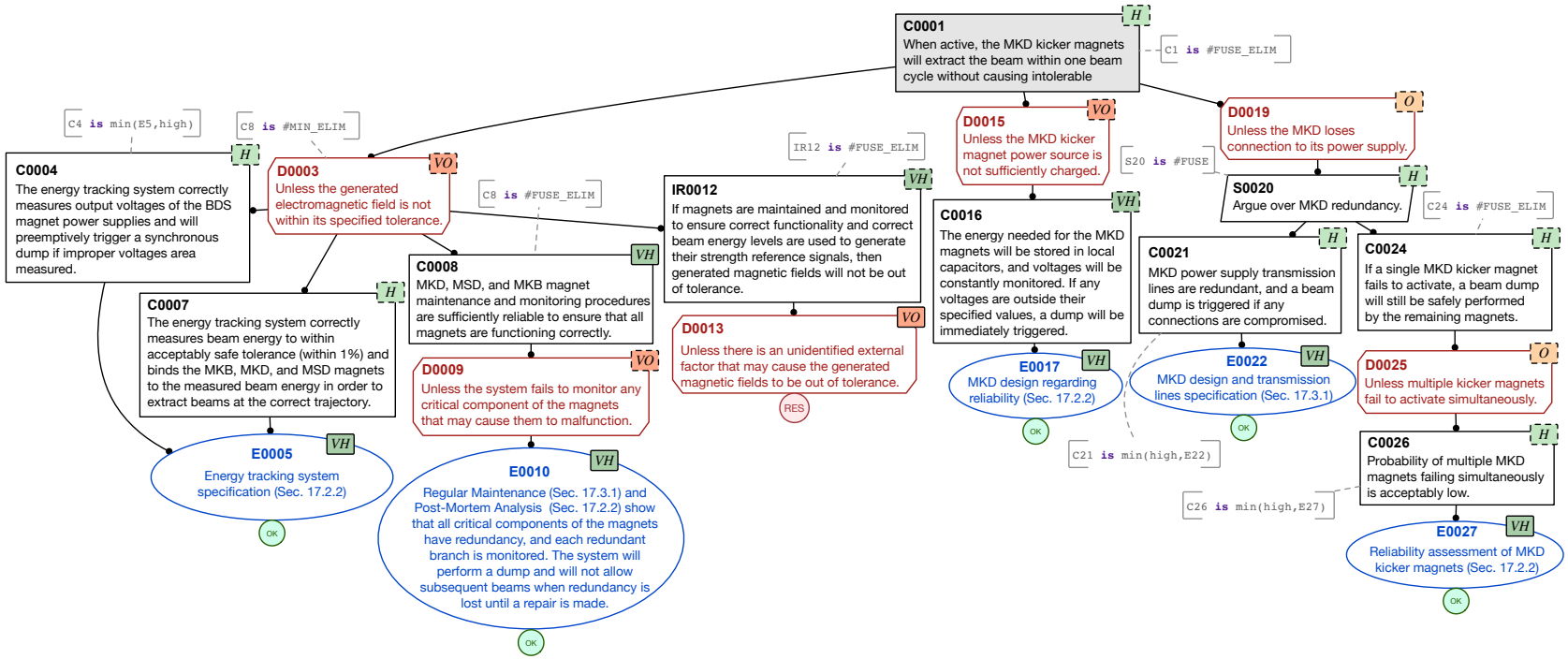


Figure D.13: Expressivity analysis result for the CERN-5 argument fragment [91].

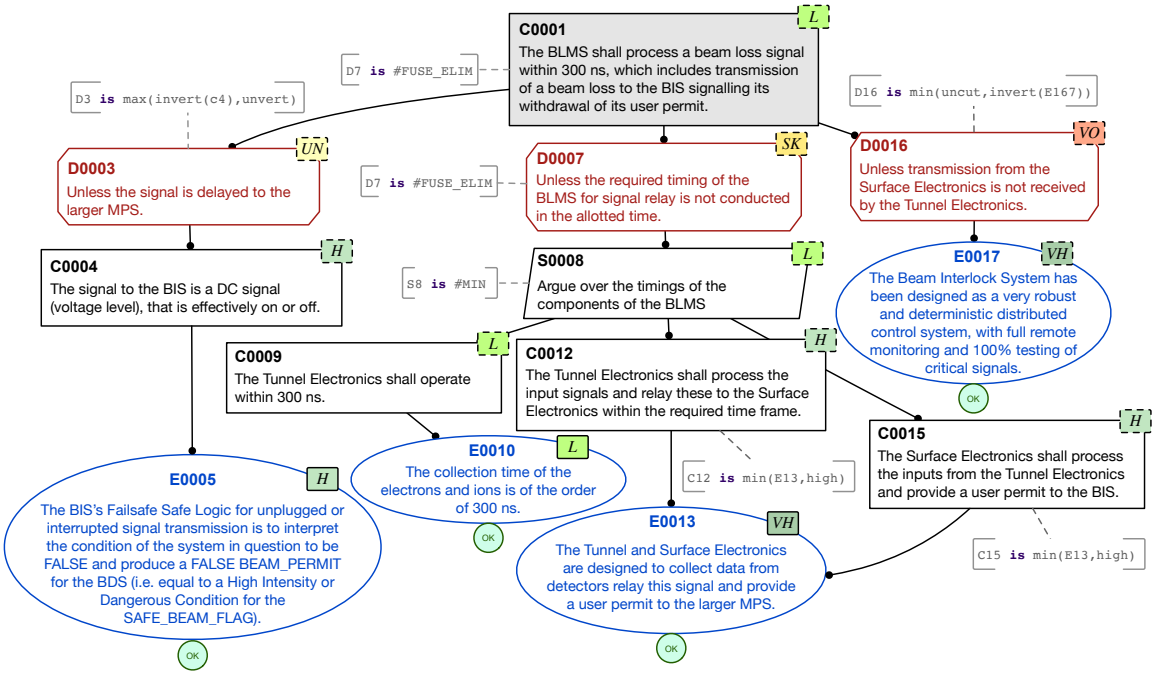


Figure D.14: Expressivity analysis result for the CERN-6 argument fragment [91].

Appendix E
Argument for the *gVent* Case Study

This appendix contains the detailed argument for the *gVent* case study from Section 7.5. The DAC's argument is divided into fragments that presented in (approximately) breadth-first order.

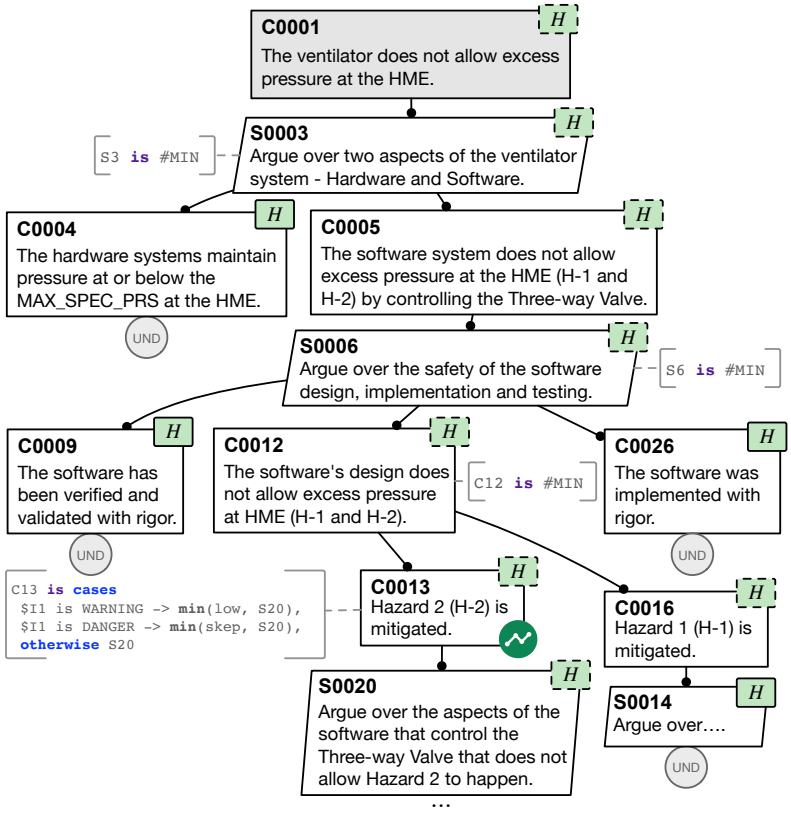


Figure E.1: Case study argument fragment for `C1`.

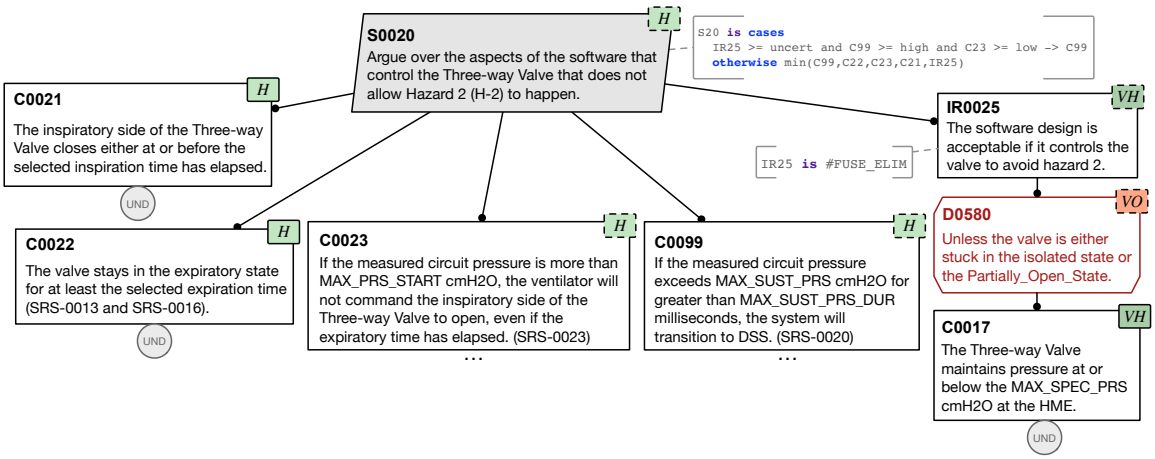


Figure E.2: Case study argument fragment for S20.

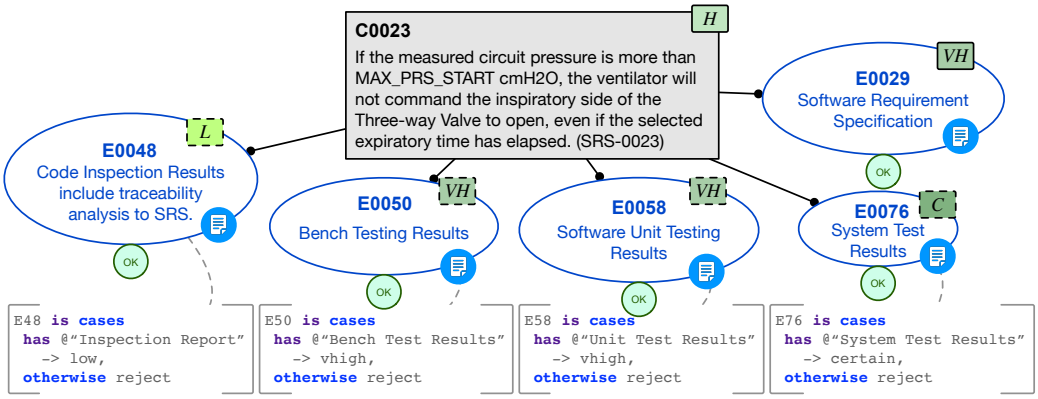


Figure E.3: Case study argument fragment for C23.

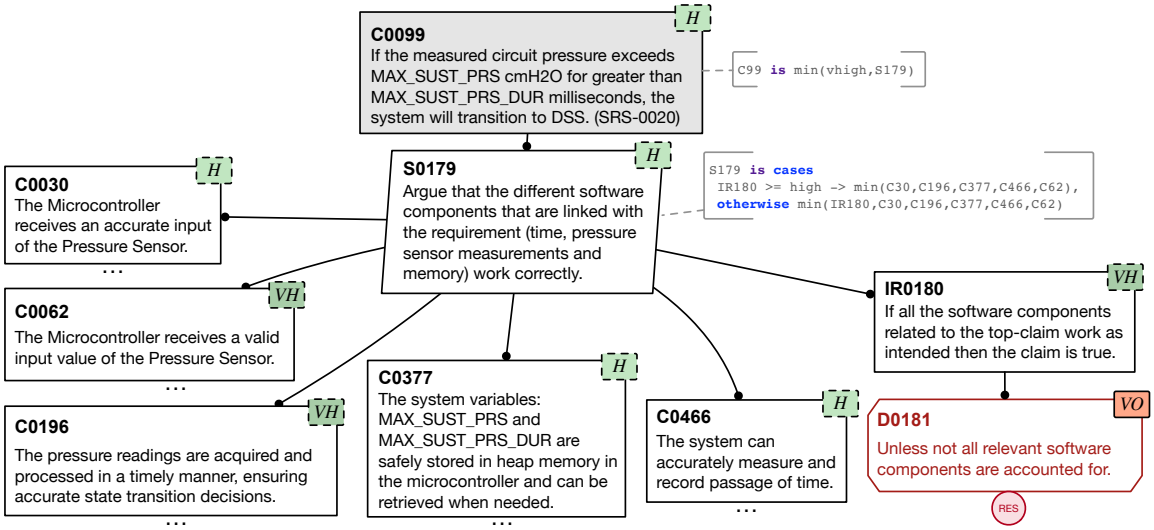


Figure E.4: Case study argument fragment for C99.

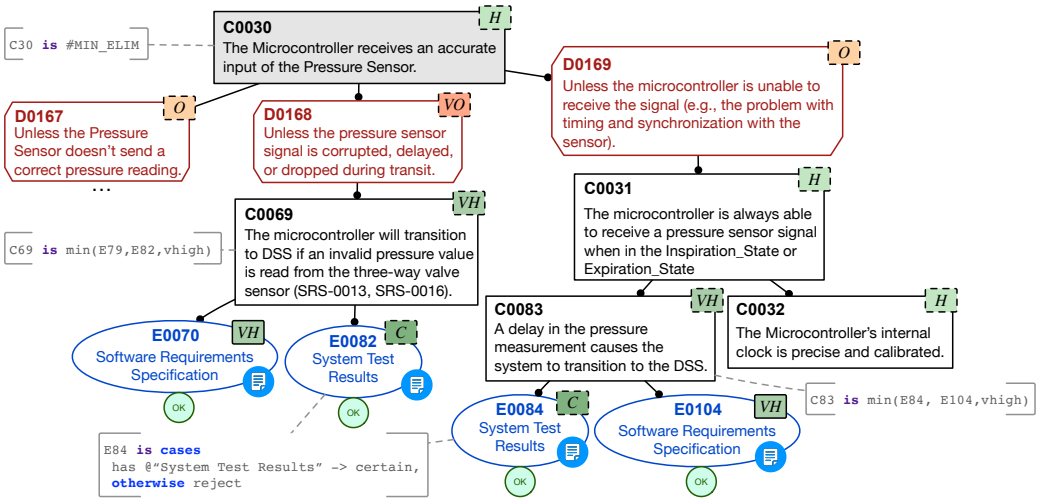


Figure E.5: Case study argument fragment for C30.

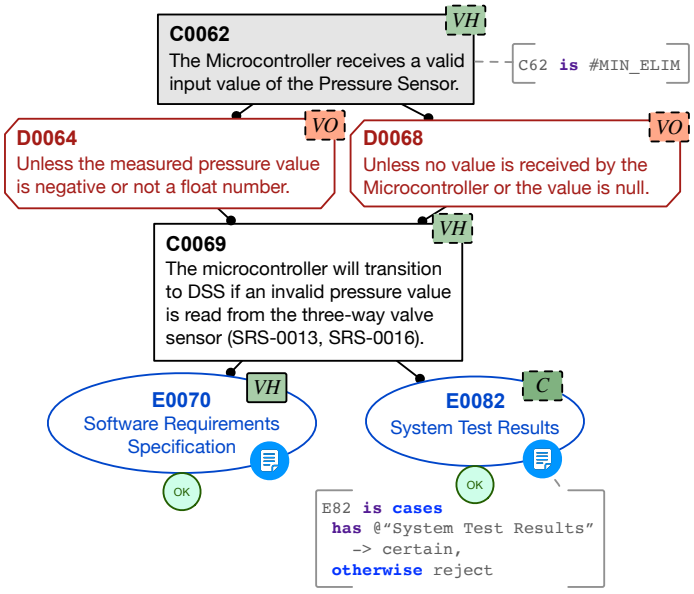


Figure E.6: Case study argument fragment for C62.

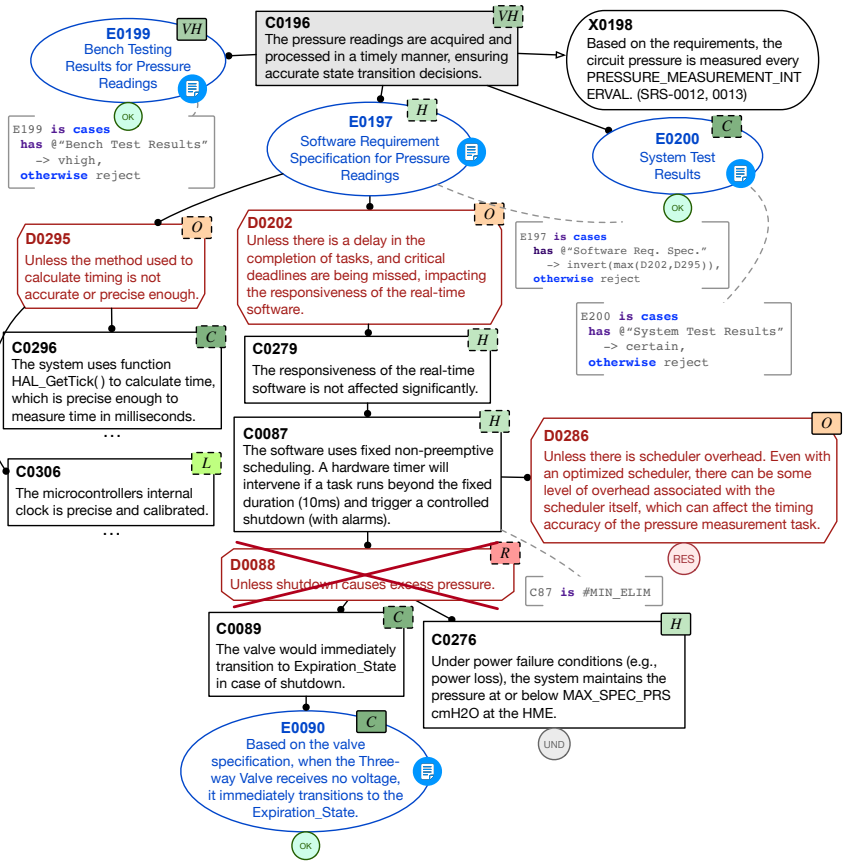


Figure E.7: Case study argument fragment for C196.

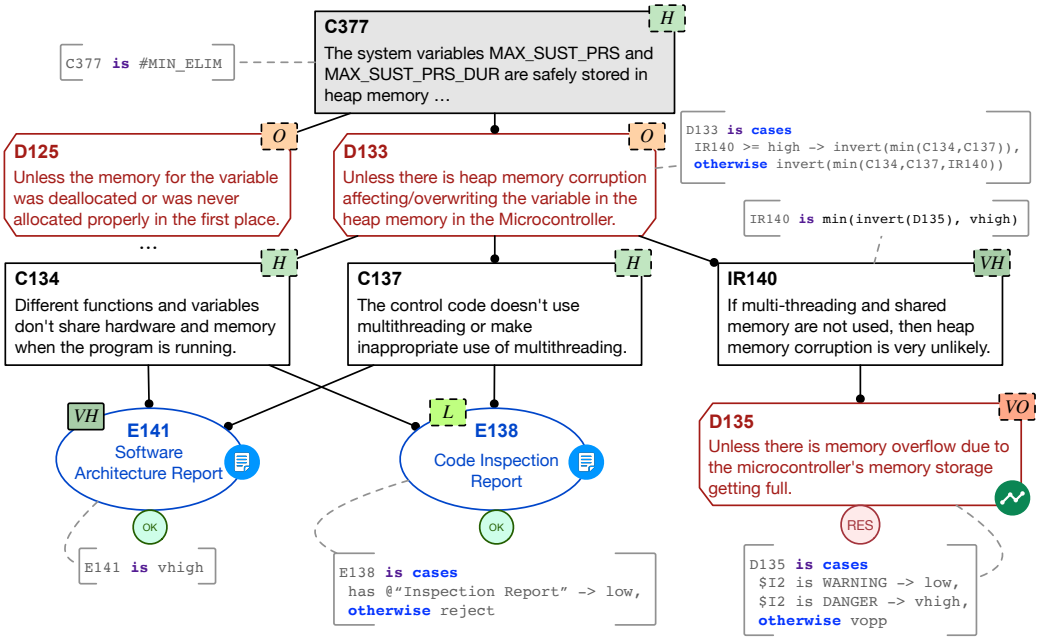


Figure E.8: Case study argument fragment for C377.

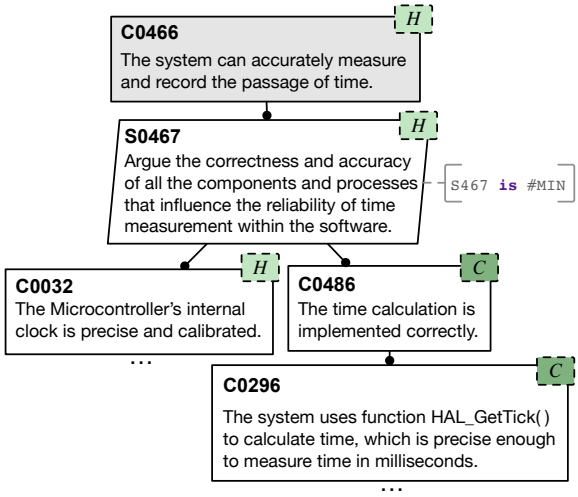


Figure E.9: Case study argument fragment for C466.

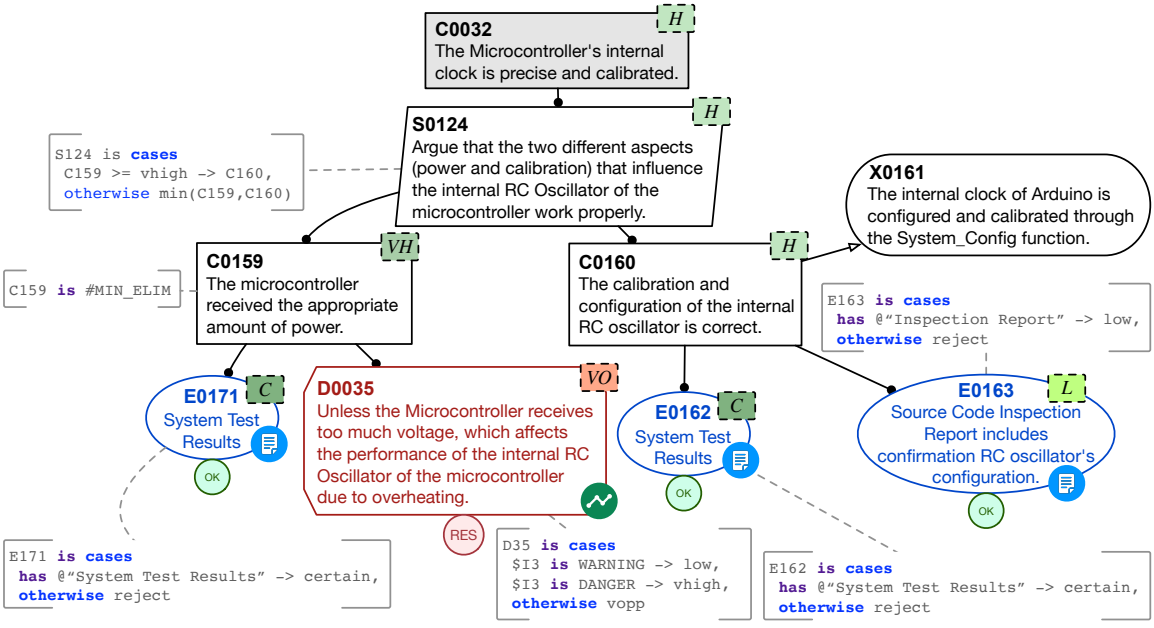


Figure E.10: Case study argument fragment for C32.

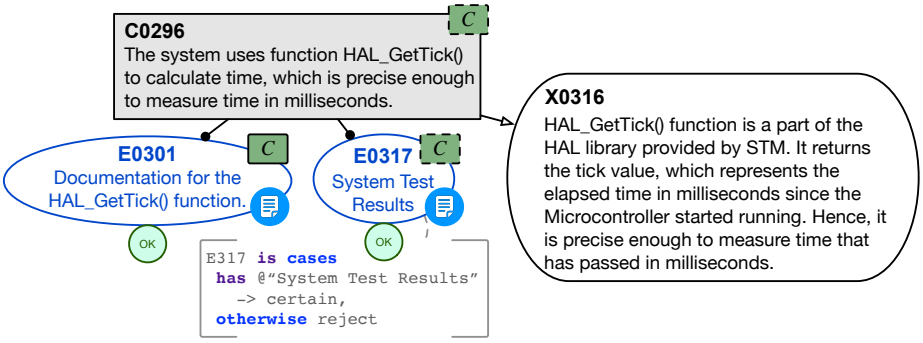


Figure E.11: Case study argument fragment for C296.

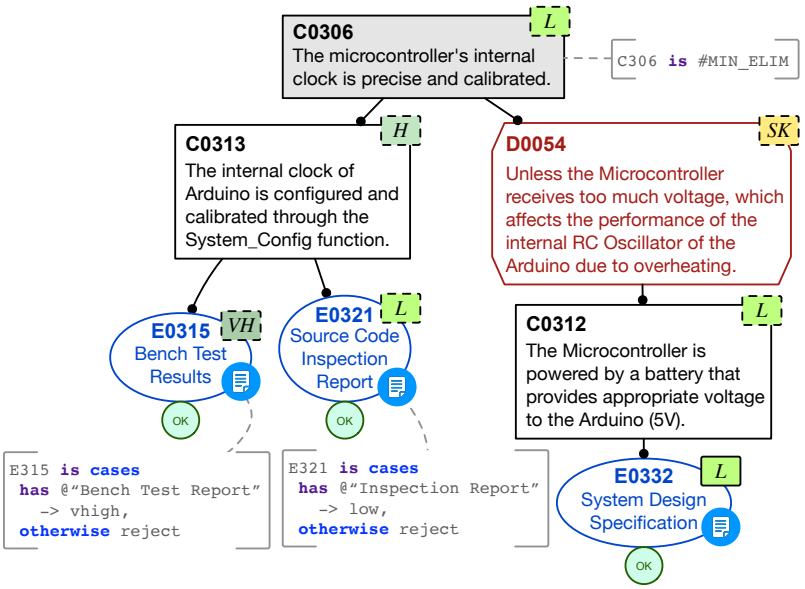


Figure E.12: Case study argument fragment for C306.

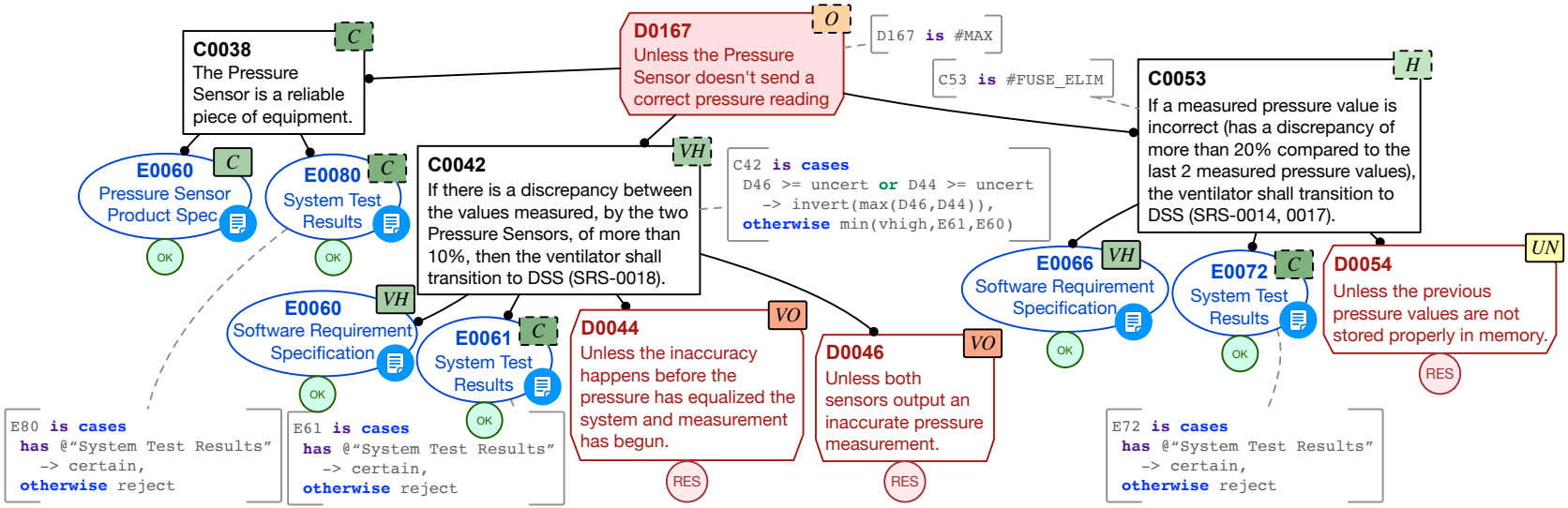


Figure E.13: Case study argument fragment for D167.

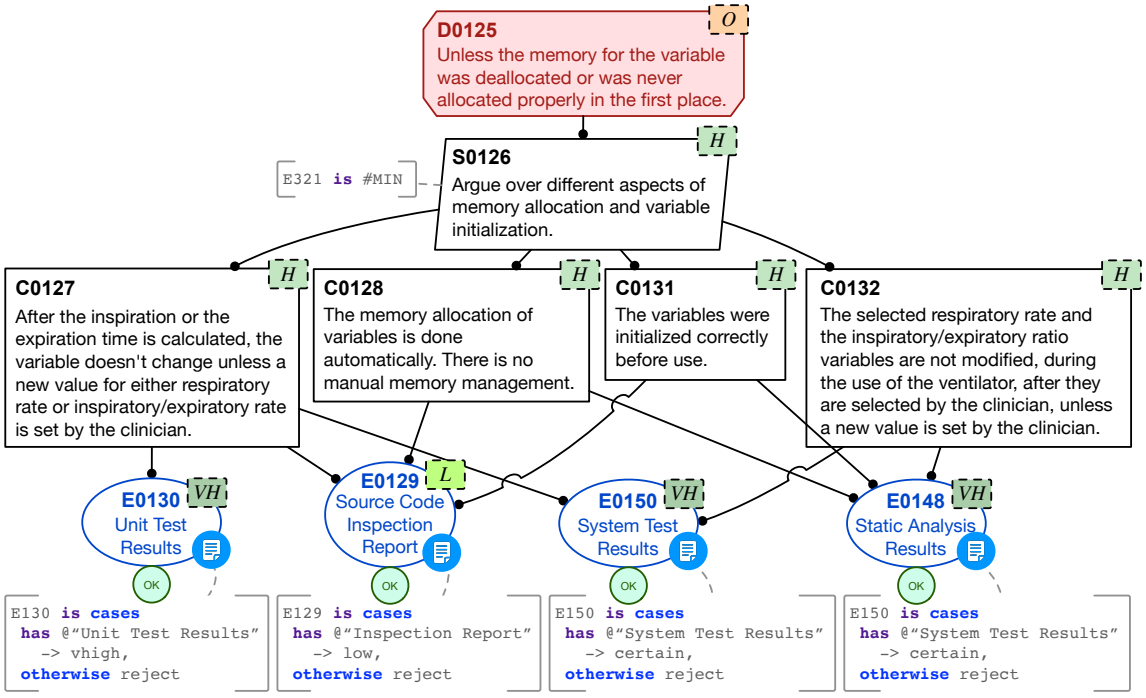


Figure E.14: Case study argument fragment for D125.