AFRL-RI-RS-TR-2022-153

ASSURANCE CASES FOR CERTIFICATION OF EFFICIENTLY LEARNED EVALUATED RAPIDLY AND AUTOMATICALLY USING THEORY OF EVIDENCE (ACCELERATE)

JOHNS HOPKINS UNIVERSITY APPLIED PHYSICS LABORATORY

NOVEMBER 2022

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2022-153 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. MCKEEVER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing and Communications
Division, Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
NOVEMBER 2022	FINAL TECHNICAL REPORT	APRIL 2020	JUNE 2022

4. TITLE AND SUBTITLE

ASSURANCE CASES FOR CERTIFICATION OF EFFICIENTLY LEARNED EVALUATED RAPIDLY AND AUTOMATICALLY USING THEORY OF EVIDENCE (ACCELERATE)

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-20-C-0515	N/A	62303E

5d. PROJECT NUMBER	5e. TASK NUMBER	5f. WORK UNIT NUMBER
		R2XM

6. AUTHOR(S)

Ken Schmidt

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Road
Laurel Maryland 20723

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	AFRL/ RI	AFRL-RI-RS-TR-2022-153

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Assurance Cases for Certification of Efficiently Learned Evaluated Rapidly and Automatically using Theory of Evidence (ACCELERATE) project was part of the DARPA Automated Rapid Certification of Software (ARCOS) Program. The ACCELERATE project completed 20 months of a 48month development effort. During this period, we focused on the design and implementation of a logic for evaluating assurance cases, as well as on the development of an initial prototype for the construction, scoring, and browsing of assurance cases by certifiers. In the initial assessment of the ACCELERATE project, we demonstrated prototype implementations of each individual module of ACCELERATE: the logic module, an argument template library, a module for the automatic construction of assurance cases from argument templates, and a module to aid in the comprehensibility of assurance cases. After this initial implementation, the ACCELERATE team focused development around a motivating example: a geofence cage ceiling monitor assurance case (“geofence assurance case”). This open source example allowed our team to more concretely define the format of templates, how they should be instantiated into assurance cases, and the correct scoring of the result with the logic module of ACCELERATE.

15. SUBJECT TERMS

Assurance case; software certification; assurance case template; Goal Structured Notation; automated software certification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	42

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
WILLIAM E. MCKEEVER	**N/A**

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 SUMMARY

This document provides an overview of the final accomplishments of the Assurance Cases for Certification of Efficiently Learned Evaluated Rapidly and Automatically using Theory of Evidence (ACCELERATE) project, which was part of the DARPA Automated Rapid Certification of Software (ARCOS) Program. As a Technical Area 3 (TA3) performer, the ACCELERATE team was tasked with creating and evaluating assurance arguments from a curated evidence store produced by TA1 and TA2 teams. The ACCELERATE team was led by the Johns Hopkins University Applied Physics Laboratory, with subcontracts to Duke University, HRL Laboratories, LLC., Siemens Corporation, and the University of Pennsylvania.

In the ACCELERATE project, we completed 20 months of a 48-month development effort. During this period, we focused on the design and implementation of a logic for evaluating assurance cases, as well as on the development of an initial prototype for the construction, scoring, and browsing of assurance cases by certifiers. In the initial assessment of the ACCELERATE project, we demonstrated prototype implementations of each individual module of ACCELERATE: the logic module, an argument template library (ATL), a module for the automatic construction of assurance cases from argument templates (A4), and a module to aid in the comprehensibility of assurance cases. After this initial implementation, the ACCELERATE team focused development around a motivating example: a geofence cage ceiling monitor assurance case ("geofence assurance case"). This open source example allowed our team to more concretely define the format of templates, how they should be instantiated into assurance cases, and the correct scoring of the result with the logic module of ACCELERATE. In the remainder of this project, our team used this motivating example to bring all modules together to form ACCELERATE, and delivered ACCELERATE v1.2.1, the geofence assurance case, and an example assurance case created for the Boeing target system, the AH-64D Apache Longbow Attack Helicopter, for the Phase 1 final assessment.

Portions of our research has been published (or accepted for publication) and presented at external, refereed venues [3, 8]. We have also successfully transitioned ACCELERATE tools and techniques to other government organizations, within the Department of Energy (DOE) and the Air Force Nuclear Weapons Center (AFNWC).

2.0 INTRODUCTION

The overarching objective of the ACCELERATE effort is to research, develop, and demonstrate an automated system that produces rigorous software certification artifacts and an accurate determination of the level of assurance. As a TA3 performer, our team is focused on the research and development of technology to automate the generation of assurance cases from curated evidence. There are two major thrusts to this effort: (1) to develop technology for the automatic construction of assurance cases for certification criteria and obligations imposed by evaluated components, and (2) to develop trustworthy technology for assessing the confidence of an assurance case argument.

The ACCELERATE proposal cites four key innovations: (I1) Rigorous definition of a novel variant of defeasible logic for representing assurance cases, (I2) natural language processing (NLP) techniques to abstract certification documents into a library of Expressive argument templates, (I3) advances to clustering, graph-matching, and transfer learning algorithms to ensure Efficient instantiation of argument templates with evidence, and (I4) improvements to interpretable machine learning to reveal Comprehensible associations between evidence and arguments.

The ACCELERATE proposed architecture comprised four modules, each mapping to one of the innovative claims. In addition to the four proposed modules, it quickly became evident that ACCELERATE should include an interface for viewing assurance cases and managing templates. This need resulted in the addition of a fifth module, the ACCELERATE Argument Browser (A2B). The full set of ACCELERATE modules include:

1. ACCELERATE Logic Library - a Python library for scoring assurance cases that integrates confidence evaluation with incomplete argumentation over assurance claims. The input to the library will be an assurance case in a graph format, and the resulting score output will be a probability density function (PDF). The Logic Library will also be capable of scoring of subcomponents and subgraphs within the overall assurance case, in order to give insight into the relative strength of sub-arguments within an assurance case.
2. Argument Template Library (ATL) - a tool that will supply a graph database of assurance argument templates, populated by prior experience and assurance arguments from literature, certification documents, requirements documents, etc. The ATL will be able to select and return all applicable argument template(s) for a given assurance case. The ATL will also perform Natural Language Processing (NLP) of system documents (including requirements documents, etc.) in order to produce new templates.
3. Automated Assurance Argument Assembler (A4) - a tool that will take a high-level claim and relevant templates as input, query a curated evidence store for related evidence, query ATL for additional templates as needed, and use the results to form an assurance case for a claim. It may then request a score for the assurance case from the logic library module of ACCELERATE. The A4 will be capable of iterating on this process, making improvements to the assurance case and calling the logic to re-score the case to result in the creation of the best argument possible.
4. Argument Comprehensibility Optimizer (ACO) - a tool that will take a scored graph representation of an assurance case, perform natural language processing (NLP) and machine learning to optimize the representation of the graph, and provide a human-readable summary of the results. Summarization of the assurance case will be available at different levels of granularity,

selectable by the user. The ACO will also link summaries of the assurance case to system requirements, certification documents, and source code when available.

5. ACCELERATE Argument Browser (A2B) - a browser-based interface for the user (e.g., certifier) to use to develop or view an assurance case in ACCELERATE. The A2B will also provide the ability to view, edit, and select the templates used to build assurance cases, and view the history of templates and assurance cases. The A2B will serve as the frontend to the entire ACCELERATE prototype, pulling together output from all other modules.

The envisioned architecture of ACCELERATE is shown in *Figure 1*.

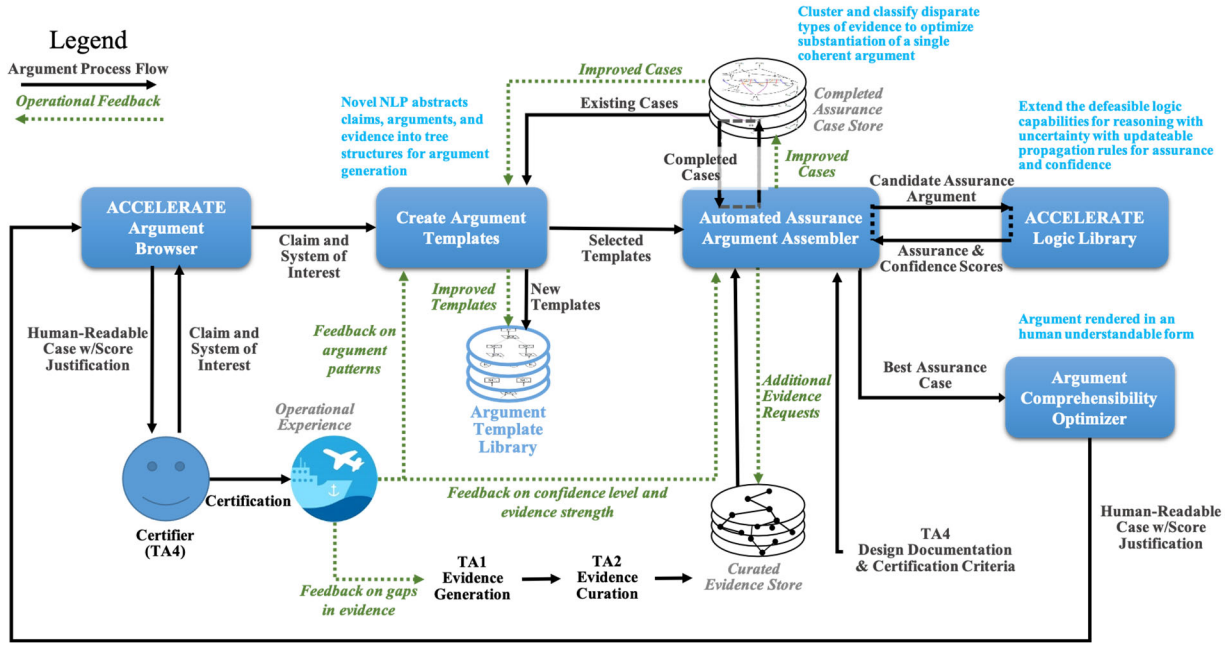


Figure 1. ACCELERATE Architecture

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The ACCELERATE project ended shortly after the ARCOS program's Phase 1 final assessment. The primary goal of the ACCELERATE team during this time was to make substantial progress on each of our innovative claims, as evidenced by the delivery of a version of ACCELERATE comprising the modules outlined in Section 2.

Our team met this overarching goal, iteratively building capability into ACCELERATE with the development and delivery of version 0.1 in advance of the initial assessment, and versions 1.0, 1.1, 1.2, and 1.2.1 delivered for the final assessment of Phase 1. The resulting tool incorporates all modules of ACCELERATE into a set of Docker containers with a single Docker Compose script and user interface, and is capable of performing a large subset of the functionality outlined in our proposal. This capability has been demonstrated on a set of use cases developed by our team. This early implementation of ACCELERATE is expected to be largely expert-driven (e.g., by certifiers, developers) and may require user interaction when testing with new inputs or use cases.

Using the A2B interface into ACCELERATE, the user is able to search for argument templates housed within the ATL, and use versioning tools to add, edit and track revisions of templates in the system (*Figure 2*). In the future, this capability will be expanded to allow user-guided NLP-based generation of new argument templates within the interface. The A2B also allows instantiation of assurance arguments using the A4 module of ACCELERATE, pulling templates from the ATL and evidence from the TA2 curated evidence store (RACK). The assurance case is then passed to the ACCELERATE Logic Library, and the resulting scores are populated within the interface for each goal and subgoal of the argument (*Figure 3*). The ACO also provides an interface within the A2B through which the user can upload source code and/or documents for the target system and produce NLP-driven similarity analysis to user-input keywords.

While the A2B is the most visible component of the ACCELERATE, each of the modules on the backend had specific technical goals and progress. The proposed and completed progress for each module is described in the subsections that follow.

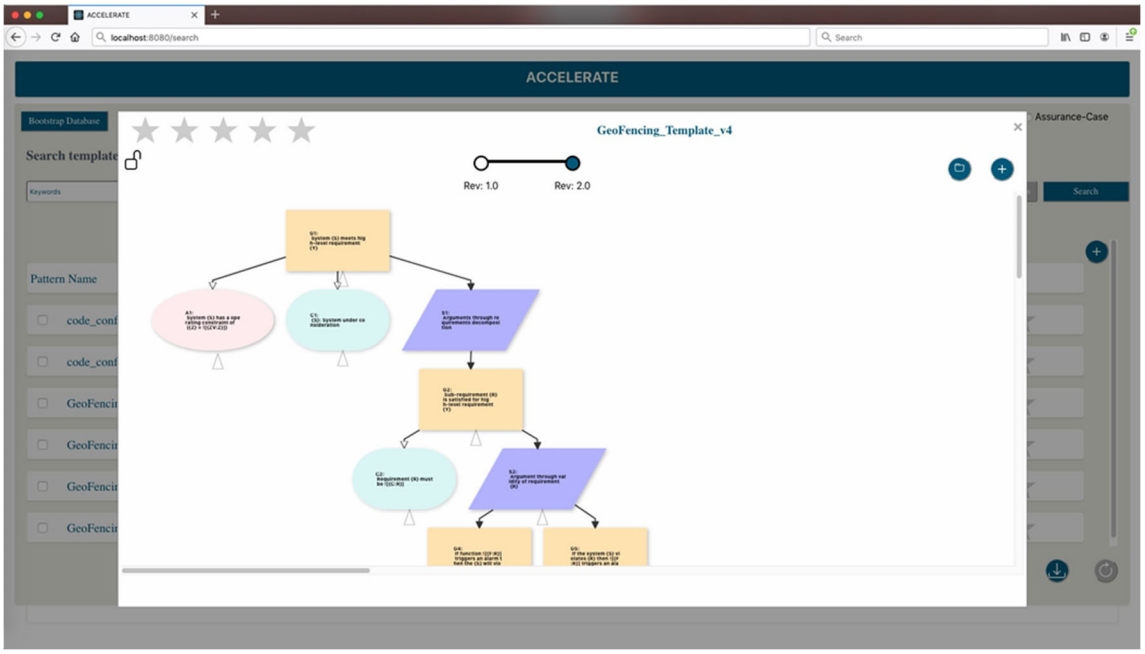


Figure 2. Argument template exploration and revision within ACCELERATE

3.1 Accelerate Logic Library

The goals proposed for the ACCELERATE Logic Library for this period of development were as follows:

1. Initial Definition of ACCELERATE Logic - develop logic tailored to establishing assurance case arguments and parameterized by abstract rules for propagating assurance and confidence levels from evidence to claims.
2. Initial Derivation of Assurance and Confidence Propagation Rules - develop propagation rules based on knowledge of specific certification domains.

Our team produced and delivered a Python module built for scoring assurance arguments, which implements the ACCELERATE logic definition and meets both of these goals. Given a Goal Structured Notation (GSN) JSON representation of an assurance argument, the Logic Library will annotate the GSN JSON with scores in the form of a probability density function (PDF) and return the resulting scored JSON assurance argument. Scores from sub-arguments/sub-goals are propagated up the hierarchy of the argument to form an overall score for the stated top-level goal.

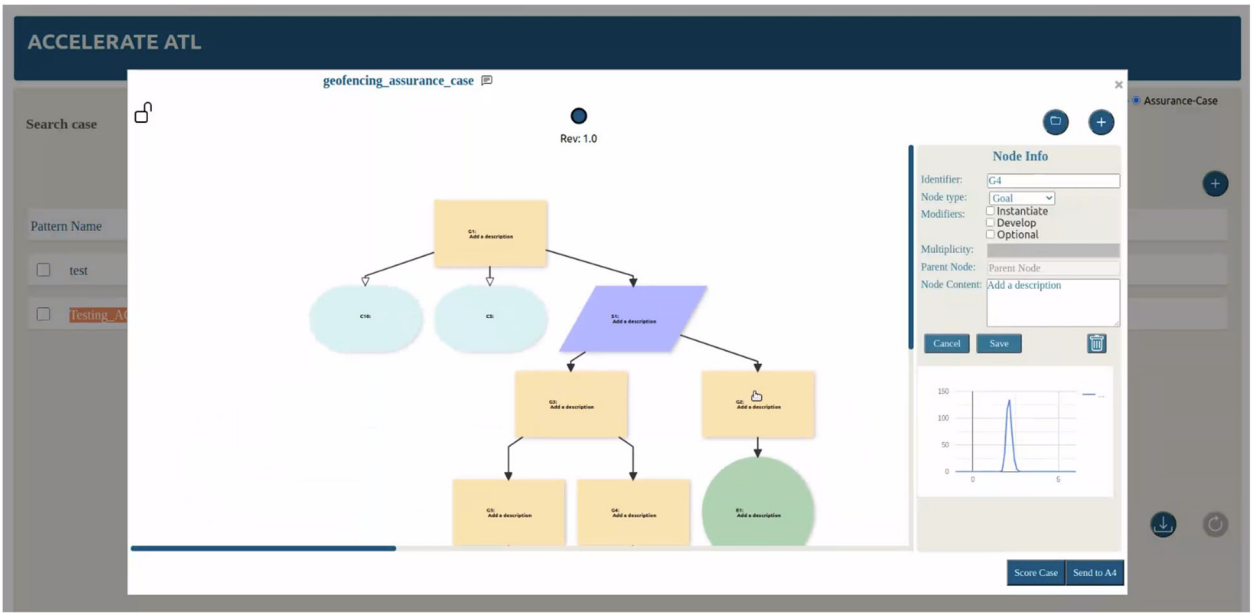


Figure 3. Assurance case instantiation and exploration within ACCELERATE

The current implementation of the library can be thought of as a "low-level" interface that conducts efficient (restricted) probabilistic inference. The library includes a wrapper that translates assurance arguments from the caller into this low-level language, from which a final PDF is computed. The Logic Library annotates the top-level claim of the GSN argument with the final PDF and returns the result to the caller. The Logic Library also recursively scores and annotates sub-arguments of the overall argument.

3.1.1 ACCELERATE Logic Implementation

The ACCELERATE Logic Library is designed to help ACCELERATE users (e.g., certifiers, developers) reason about their level of belief (confidence) as to how reliable properties about a software system are in practice (assurance); these properties depend on other "lower-level" properties, forming a property graph. The "final score" for a system is the measures of confidence and assurance for the graph's root property. This root property could, for example, represent the system's overall compliance with stated system requirements, or its compliance with certification standards.

Our work on the logic consists of developing the underlying mathematical theory (including a formal language with semantics given by efficient, restricted probabilistic inference), implementing this language as a Python library (intended to serve as the "low-level" interface), and beginning work on the "higher-level" interface, which will score assurance cases produced by A4. Although we plan to automate as much of the certification process as possible, we expect that estimating the reliability of software, in practice, will require judgment calls informed by expert elicitations. We have designed the logic for efficient inference, enabling interactive usage.

We note that many aspects of automating software certification remain open research questions. While we believe that we have identified a suitable approach, we have also made strong efforts to produce a highly modular and flexible logic and implementation that can be adapted to other approaches. We describe the underlying theory, implementation, and future plans for automated scoring of assurance cases in the following subsections.

Theory The ACCELERATE Logic is designed to efficiently reason about properties of a given software system. We expect these properties to be predicates, i.e., binary / Boolean statements about (sub)components of the system. We inform our judgments with evidence, a catch-all category for any information that makes us more or less confident about how well a particular property will hold, in practice.

A key concept is that each probability distribution associated with some property represents a user's level of belief about how well that property will hold in practice, as the system is expected to be deployed. We note that this belief about properties will depend on beliefs about other properties. As a simple example, "Does this control algorithm successfully keep an aircraft stable?" could depend on "Does this linear algebra library correctly compute this coordinate transformation?", the properties and their dependencies form a directed acyclic property graph.

Ideally (from a certifier's perspective), we would directly prove the correctness of some software system using end-to-end formal verification. In other words, all necessary properties would be written as machine-checkable specification, the code would be accompanied by formal, machine-checkable proofs of correctness, and the certification process would consist of simply verifying that the formal specifications matched the human-readable (informal) requirements and that the proofs executed correctly.

In practice, formal methods tools and techniques do not yet scale to the size of systems that are often deployed and, due to time and expense, formal methods are only suitable for small, mission-critical subcomponents. In addition, a large amount of informal evidence, informed by expert judgment, is brought to bear in the certification process. For example, the mere existence of an informal (not machine-checkable) specification as to how software should work often gives users a higher degree of confidence that the software works correctly - because the existence of such a specification is evidence that the developers performed non-trivial effort in reasoning about how their program should operate (Lamport et al, 2018). (We note that mandating such informal specifications may inadvertently destroy much of their value, as a consequence of Goodhart's law (Goodhart).

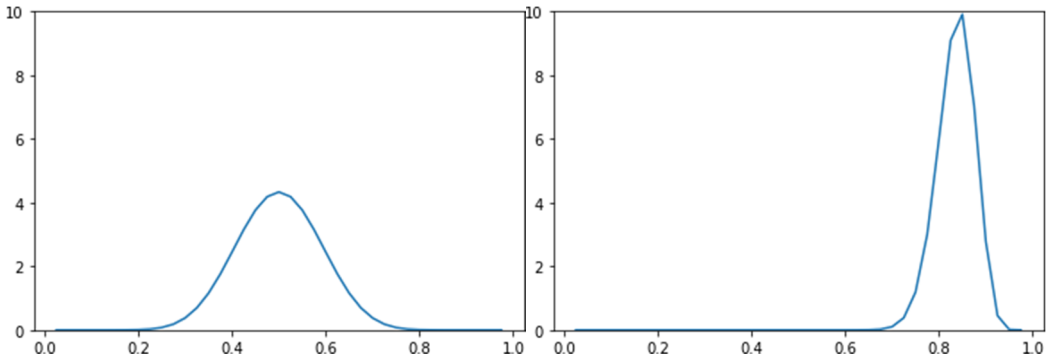
Since we expect to process heterogeneous, often informal evidence, the semantics of the ACCELERATE logic are built on top of probability theory. A probability distribution, with support over [0, 1] represents a user's level of belief as to how often the property holds in practice; in the extreme case, perfect knowledge that property holds perfectly would be represented by a Dirac-delta distribution centered at 1. Examples of more representative cases are given in *Figure 4*.

The logic is built upon the 'primitives' of probability distributions and operators for combining distributions. For distributions X and Y , with support [0, 1], we define and(X, Y), not(X), or(X, Y), and mix(X, Y, w_X, w_Y), and is the product distribution of X and Y; not is the distribution of $1 - X$. or is not(and(not(X), not(Y))), following De Morgan's laws, and mix is the weighted mixture distribution of X and Y .

Heterogeneous evidence can be incorporated into the system by mapping the evidence to pseudo-counts. A Beta distribution - which has support [0, 1] and is the conjugate prior distribution for Bernoulli likelihood — is parameterized by α and β, which correspond to the number of successes and failures. A particular piece of evidence (e.g., "A component passed a test via the XYZ testing methodology") needs to be mapped to the number of expected successes and failures that this would yield in practice. Under ideal circumstances, this would be fully data-driven, based on past experience (e.g., "based on past projects, similar components that passed this test operated correctly n times and failed *m* times"); we expect that, in practice, these updates will rely heavily on

expert elicitation. Bayesian updates are remarkably straightforward, given Bernoulli likelihoods and Beta priors; the posterior (hyper) parameters are simply $\alpha + \Sigma x_i$ and $\beta + n - \Sigma x_i$ (Jaynes, 2003).

The operators can be interpreted as follows: and is the resulting distribution when both properties must hold. or is the distribution when either property is sufficient. not is the distribution of the property failing. mix can be used to represent probabilistic dependence: if something depends on a property some fraction of the time w, then we can express this probabilistic dependence as and(X, mix(Y , δ_1, w, $1 - w$)) where δ_1 is a Dirac-delta distribution centered at 1; note that Beta(1, 0) is equivalent to δ_1.



(a) Low confidence, low assurance (b) Higher confidence, higher assurance

Figure 4. For a given property, a probability with support [0, 1] represents a user's level of belief as to the assurance of that property. Informally, the x-axis represents how reliable the (sub)component of the system is and the y-axis represents how strongly that belief is held

Efficient inference A critical problem with the naive definition of these operators is that they are not closed, e.g., the product distribution of two Beta distributions (the and operator) does not, in general, produce another Beta distribution. This, in turn, affects our ability to perform fast Bayesian updates. With conjugate priors, incorporating evidence is simple arithmetic; with more general classes of probability distributions, updates would require computationally expensive numerical methods, such as a Monte-Carlo Markov Chain. Computationally expensive updates would make interactive usage impractical.

We recover efficient inference by making one modification to the definition of the operators: the result of each operator is the Beta distribution that has the same first and second moments (mean and variance) as the probability distribution resulting from the naive definition. In other words, we find the best "fit" Beta distribution, that matches the original definitions of and, or, not, and mix.

Our original research contributions here are *closed-form* expressions that let us compute these distributions efficiently (the derivations also act as a proof that the resulting distributions are, in fact, unique). The closed-form expressions are complex and were derived with assistance from a computer algebra system. The full derivation is included in the accel package.

3.2 Argument Template Library (ATL)

The goals of the ATL module of ACCELERATE for this period of development were:

1. Initial Extraction of Claims and Evidence from Legacy Certification Documents - develop NLP techniques for extracting the evidence and claims structures from legacy certification documents.
2. Initial Template Library Implementation - service and interface for storage, query, retrieval, and optimization of argument templates.

These goals were exceeded by our team during the ATL module's development. The ATL is packaged as a set of Docker containers that provides a pair of Neo4j databases, one for storing argument templates and one for storing assurance arguments, and a REST API that can be used to query the ATL for relevant argument templates. The REST API provides calls to allow the user to list all of the templates currently in the ATL's database and request a specific template by its unique ID. The REST API also supports an initial implementation of the ATL's NLP component, through which the user can retrieve argument templates that have a similarity to a user-specified keyword or phrase. While all of the argument templates currently present in the ATL are human- generated, the ATL also provides an initial implementation of the ability to create new templates from requirements documents. The ATL also provides the core database for the ACCELERATE system, housing and maintaining version information for both templates and assurance arguments in Neo4j instances. Templates present in the ATL provide a framework for various types of arguments, as well as defeaters to arguments.

Through its REST interface, the ATL integrates with both the A2B and the A4. The ATL containers also host the ACCELERATE Logic Library Module, which the ATL uses to score argument templates and assurance arguments hosted in its Neo4j database, and the A4 module.

3.2.1 ATL Implementation

The Argument Template Library (ATL) provides a set of argument templates that can be used by the Automated Assurance Argument Assembler (A4) module to automatically construct assurance cases. Argument templates use a domain specific language based on the Goal-structuring notation (GSN). The argument templates currently stored in the ATL are curated from multiple source - certification documents, public domain documents, academic publications, etc., and are adapted to the safety use case under consideration.

Our team developed a baseline set of argument templates that are pre-loaded into the ACCELERATE ATL database. Example templates in the current set include "Requirements Decomposition", "Requirements to Hazard Mapping", and "Test Coverage for Requirement". These were produced manually from literature and from DO-178C. Argument templates can be composed by the A4 into larger arguments for an overall system (*Figure 5*).

The ATL relies on the use of GSN patterns to standardize the representation of argument templates for use by other modules of the ACCELERATE framework. The purpose of the ATL is to; first, serve the stored GSN patterns and instances to the A4 module; second, to extract information from the specification, certification and assurance case documents for storage and creation of new argument templates; and, finally, to generate and/or extract query specific argument templates for use in other ACCELERATE modules.

The ATL module also manages the interface between ACCELERATE and the storage databases for both argument templates and assurance cases. To support this, the ATL introduces a graph versioning concept that permits a user to version argument templates and assurance cases in order to maintain ability to track changes made to these data structures.

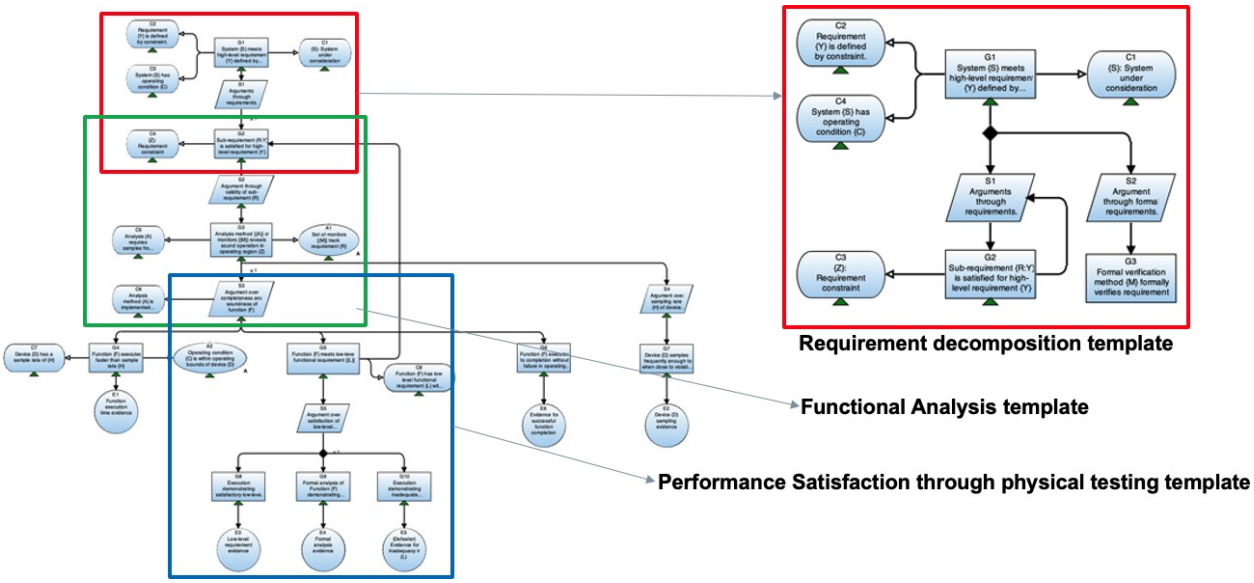


Figure 5. Composition of three argument templates from the ATL into a larger assurance argument

ATL Architecture The ATL is implemented as a web server serving REST endpoints that permit the user to modify the content stored in the database. Through these endpoints, users can:

- Upload argument templates and assurance cases for storage in the database (acceptable formats include JSON and the AdvoCATE export format),
- Edit argument templates and assurance cases by modifying the attributes or content of the template,
- Create new revisions of the uploaded content, and
- Score assurance cases and retrieve the confidence scoring results.

The working architecture for the ATL is illustrated in *Figure 6*. The ATL is instantiated by an application loader context that starts a Flask application. The Flask application manages the micro-services that operate the NLP, GSN and data services. The data and GSN services are the most developed in ATL v1.0, while basic functionality is provided in the NLP service. These services are triggered by the requests made through the REST API. The document ingest API is implemented in anticipation of future development of the ATL, which will include parsing documents to extract information of interest to the certification process.

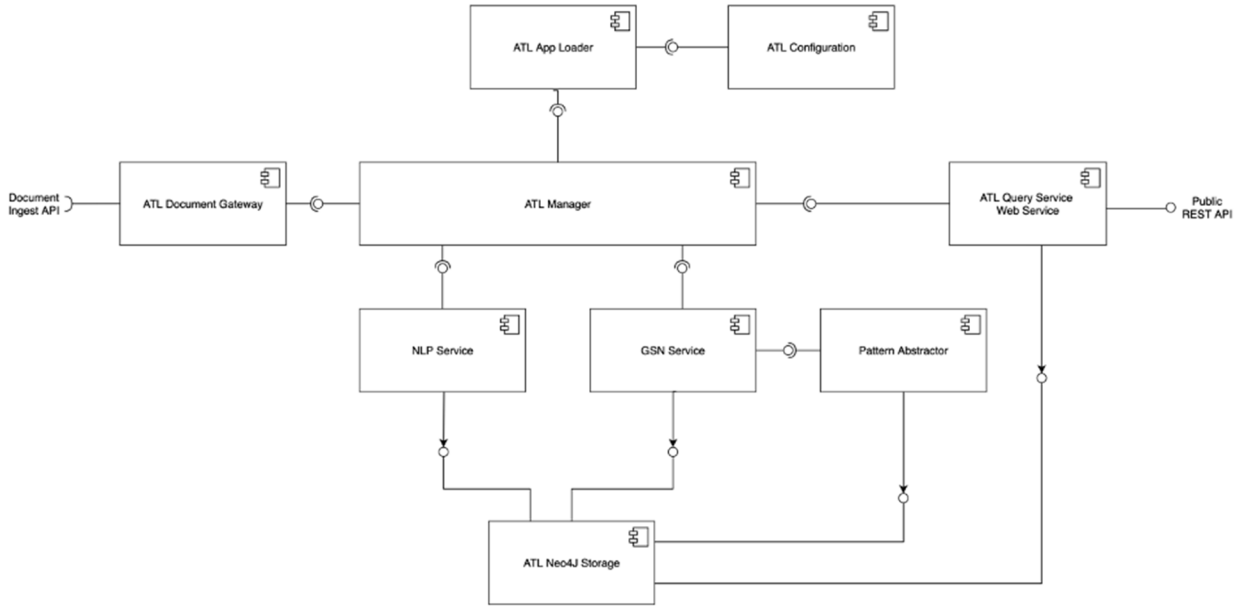


Figure 6. System architecture of the ATL

Argument Template Syntax An argument template/assurance case consists of:

- A set of nodes, with one of them being designated the root node,
- A set of directed edges forming a directed graph, with all the nodes reachable from the root node (Cycles are allowed in templates, but not in assurance cases.),
- A set of node attributes for each node,
- A set of edge attributes for each edge (edge attributes are only used in argument templates; assurance cases do not have edge attributes)
- A top-level metadata dictionary, for global metadata that applies to the entire template/assurance case.

Both argument templates and assurance cases include the following node attributes:

- 'content': (string, required) contains the main content of the assurance case, i.e., this is the claim or goal for the assurance case, while all other attributes are effectively metadata for the assurance case,
- 'class': (string, required) the type of node represented by the object, e.g., 'goal', 'strategy', 'solution', 'justification', 'assumption', 'context' or 'choice' (A key feature of argument templates is that they can include choice nodes. The 'content' of the 'choice' nodes must have the form *m of n*, where *m* and *n* are concrete numbers. *n* must be equal to the number of children the node has, and *m* must be between 1 and $n - 1$ (inclusively). The choice node indicates that at least *m* out of *n* branches must by developed in order for the assurance case to be valid.),
- 'identifier': unique ID for the template/assurance case,
- 'operator': the logic operator that should be applied to the node when computing confidence of the node based on its children (this is only valid for goal and strategy nodes),
- 'is \defeater': Boolean indicator identifying whether the node is a defeater node (if the node is a defeater, this triggers the negation of any children nodes).

The following node attributes are specific to argument templates. These are required for accurate construction of assurance cases by the A4.

- 'modifiers': (string list) a list of modifiers to be applied to the template node during instantiation. Valid modifiers include:
  - 'instantiate': indicates that the content of the node has a placeholder variable that has to be instantiated.
  - 'develop': indicates that the template is incomplete and needs to be developed by linking to another template. Valid children templates are identified in the metadata of the parent template (in the children key, indexed by the node's identifier).
- 'children': (list of nodes) children nodes of the current node

The following are the edge attributes stored between the parent and child nodes in the template:

- 'constraint': (string) a python expression that imposes a constraint on the existence of the edge. If the python expression is evaluated to be true, the edge exists, otherwise the edge should not be added to the assurance case.
- 'multiplicity': (string) the number of times a child node can be instantiated and the parameter defining the number of occurrences of the child node. The multiplicity follows the syntax *m : X : Y, Z* where *m* is the variable defining the number of occurrences of the instance, *X* is the base parameter which is repeated m number of times, *Y, Z* are the parent parameters from which the parameter *X* is derived.
- 'is optional': (Boolean) a flag representing whether a node is optional in the instantiation of the assurance case.

Argument templates are also required to have the following top-level metadata:

- 'name': (string) a unique name for the template. This name must be unique across all the templates stored in the database.
- 'children': (dictionary) specifies the inter-template mapping in the database (possible valid child templates for the current template).
- 'placeholders': (dictionary) specifies a mapping of the placeholder variable to the placeholder description (e.g., *S = System*, *Y = high level requirement*, etc.) for all the placeholder variables that are used in the template.

Graph Versioning ATL implements a time-based versioning of graphs stored in its databases (argument templates and assurance cases). The time-based versioning implementation separates the object from its state and uses relationship properties of the graph format to record changes. This concept is illustrated in *Figure 7*, where the assurance case is versioned by duplicating its content across multiple revisions. The revisions themselves keep track of the history of changes that are performed by the user. This allows any past state to be revisited at any point in the future.

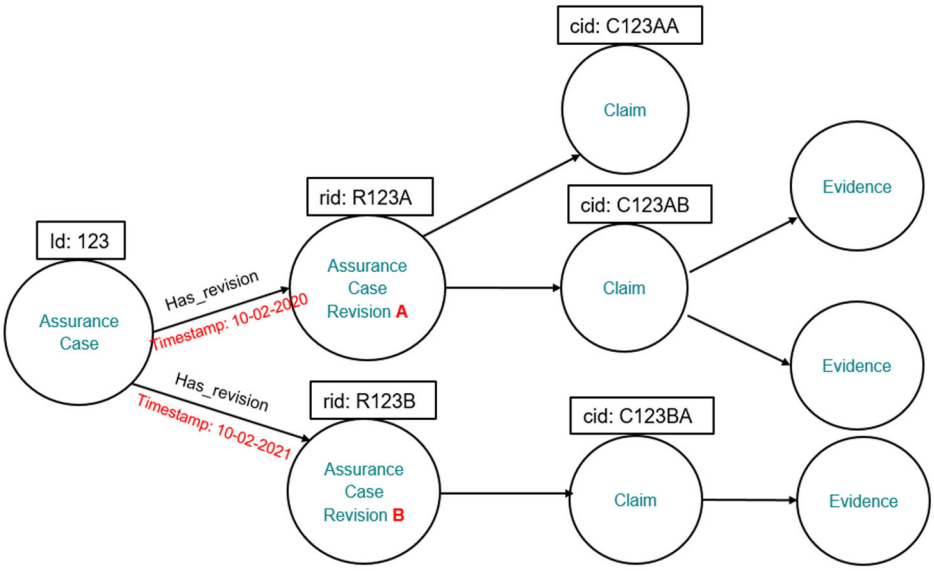


Figure 7. Time-based versioning of graphs

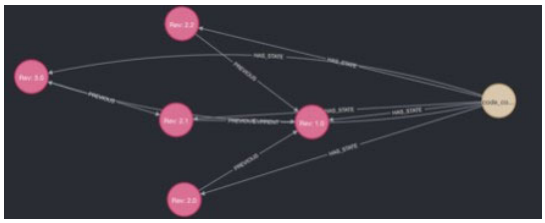


Figure 8. Realization of multiple revisions of a template in the database

An example of revisions is illustrated in *Figure 8*. As seen in the figure, the revisions can be traced back to their source and multiple branches can be created from any revision. Each revision of the argument template/assurance case can also be traced back to its source, as illustrated in *Figure 9*.

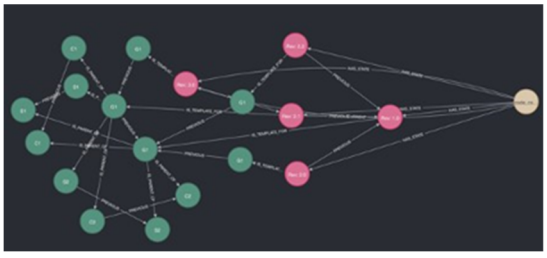


Figure 9. An illustration of traceability of changes to the content/attributes of a template across different revisions of the template

3.3 Automatic Assurance Argument Assembler (A4)

The A4 module development focused on the following tasks during this period of development:

1. Initial template pattern and evidence clustering - develop methods for clustering template patterns and evidence to learn the relationship between arguments, evidence, and assurance case topics.
2. Initial Argument Instantiation - develop methods for argument instantiation to determine the argument patterns and pieces of evidence that are most appropriate for a given assurance case.
3. Initial Substantiation Completion and Optimization - develop methods for creating assurance argument candidates by combining the selected argument templates and evidence.

The A4 module delivered with the latest version of ACCELERATE implements the core needs of the tasking outlined. However, the goals also include clustering-based tasks that have not been completed. These tasks were initially planned under the assumption that there would be a surplus of evidence and argument templates during Phase 1 of the program, requiring the A4 to pick which were most relevant to the claim. With the current set of system evidence and argument templates, the process of selecting candidate argument templates for the instantiation of a given assurance argument is fairly straightforward. Additionally, the amount of evidence in the curated evidence store is limited, and furthermore, our initial tests in evidence clustering have shown that this approach is less necessary than initially thought. Clustering could be used as an optimization for assurance argument instantiation, but is not a core requirement for instantiation.

During the months following the Phase 1 final assessment, our team re-evaluated the role that machine learning could play in the generation of assurance arguments. As part of this, we conducted initial research and developed initial definitions of a comprehensive evaluation metric that reflects on the quality of and completeness of an assurance case. With this metric defined, we proposed to pivot the clustering tasks into a more generalized machine learning approach that would result in a more optimized assurance case. The ACCELERATE task was ended before we began execution on these proposed methods, but they remain an area of interest for future work.

However, outside of the clustering-based tasks, our team has met all of the proposed goals of the A4 module for this period of execution. The current implementation of the A4 provides a REST API through which it can receive selected templates as input for instantiation. With this input, the A4 queries the RACK for system information and evidence, and may query the ATL for additional templates as needed, then instantiates an argument. The A4 will return a JSON representation of the resulting Goal Structuring Notation (GSN) argument through the REST API upon request. The A4 also makes use of the RACK's REST interface for integration with the RACK and retrieval of evidence.

The A4 is also integrated with the Logic Library to enable incremental scoring of the assurance case. In this way, the A4 can test options when a choice is presented in an argument template, in order to instantiate the most favorable assurance case before passing the result back to the ATL for final scoring and storage. The A4 is not yet optimized, i.e., it may not use all evidence and is not guaranteed to output the best argument for a given claim. Optimization requires a way to measure the quality of an assurance case for a given system in the presence of defeaters. We explored evaluation metrics proposed in the literature to assess the overall quality of an assurance case. Our

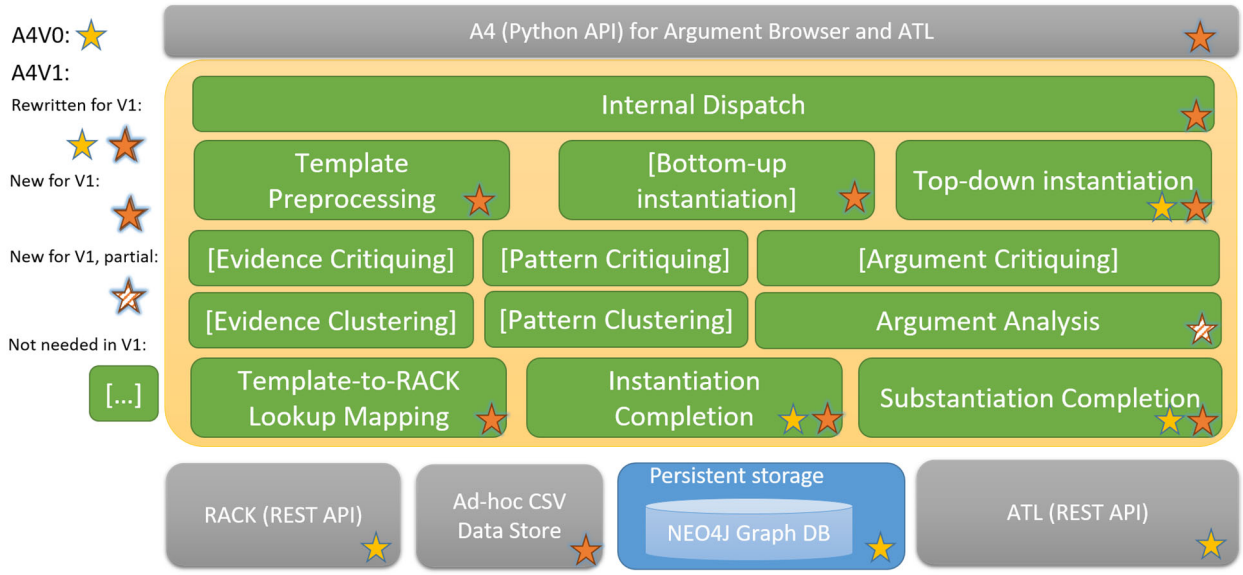


Figure 10. A4 software architecture

findings revealed evaluation metrics that are mostly subjective and lack quantification. We also explored using metrics obtained from the different modules of ACCELERATE to fill this gap and quantify quality. Section 5 details our future work plans in this area of research. We propose to integrate machine learning algorithms and novel assurance case quality metrics for optimization purposes in A4. Optimization would guarantee that the right evidence, artifacts, and defeaters are used to instantiate the best possible assurance case for a given claim.

3.3.1 A4 Implementation

The objective of the Automated Assurance Argument Assembler (A4) module is to automatically instantiate argument templates with evidence to construct the most accurate/predictive assurance argument possible. Our A4 module is conceived as a graph-based machine learning module that learns argument instantiation rules and heuristics from argument context. The module aims to augment the reasoning of system designers and certifiers to achieve accurate and scalable assurance generation processes with high fidelity and validity.

The A4 software architecture is summarized in Figure 10; components with a star annotation are part of the current implementation.

A4 v1 is not intended to be invoked by the end-users directly; it is intended to be used via the A2B interface. After A4 is invoked via A2B, the resulting assurance case is placed into the Neo4J database and the handle to the new assurance case is passed back to ATL.

A4 effectively defines the syntax and semantics of both the argument templates it consumes and of the assurance arguments it generates. The A4 source code distribution contains a detailed A4_IO_Format.md document that describes the format of the templates A4 expects, the format of the assurance cases A4 outputs, and how the former specifies the latter.

Assurance Case Instantiation Given an argument template and top-level variables, the A4 performs the following steps to build an assurance argument:

1. Verify that the argument template follows the correct syntax defined by the A4; provide early feedback to the user/ATL if there is any issue with the argument template
2. Verify that the data mapping file, which maps the structural and quantitative evidences from argument templates to data sources (e.g., RACK), is complete
3. Instantiate the argument template:

   (a) Multiply any subtrees present in the template as the "multiplicity" attribute in the template dictates. This requires using the evidence mapping file to look up structural evidence about the system (e.g., number of and identifiers for sub-requirements or sub- components within the system).

   (b) Instantiate nodes within the template with structural and quantitative evidence, mapping the latter to initial weight values for the ACCELERATE logic library to use for scoring.

   (c) Recursively execute the A4 on any nodes within the template that have a "Develop" attribute. This attribute indicates that the node should be instantiated with a child template. Once the child template is constructed and selected, it is joined with the parent node within the original template.
4. Verify that all instances in the template where there is an "m of n" requirement have an appropriate number of children nodes.
5. Verify that there are no cycles within the resulting assurance argument, and maintain state in order avoid entering instantiation cycles when recursively executing A4 for instantiation of "Develop" nodes.

3.4 Argument Comprehensibility Optimizer (ACO)

The ACO planned development during this period of execution included:

1. Data Preparation - collect representative assurance cases to construct a training dataset for machine learning.
2. Idiom Learning - develop methods in Bayesian inference that reveal recurring idioms in assurance cases.

The ACO development took a different approach than initially planned. The machine learning approach identified requires a large repository of assurance cases to construct a valid training dataset. Unable to locate or collect a large enough repository in the initial phase of development, ACO progress instead focused on the development of a capability to link assurance case keywords to underlying source code and system documents, producing a tool called the ACO Connector.

The ACO module's Connector component is realized as a set of Python scripts that perform similarity analysis, comparing keywords extracted from requirements documents and source code to a set of queries provided by the user. In later versions of ACCELERATE, this output will be used to tie requirements documents and source code to assurance arguments or claims. The ACO expects a human in the loop to load documents and source code into the tool via a web browser-based interface that is loosely integrated with the A2B, and to interpret the output.

3.4.1 ACO Implementation

The ACO has three major objectives:

1. Simplify the assurance case by layerizing and clustering the assurance case. Reduce height, width of the tree, without compromising its integrity, achieved by the *Summarizer*.
2. Connect particular arguments with particular sections of external documents (e.g., regulations, requirements), achieved by the *Connector*.
3. Connect particular arguments with related pieces of source code, also achieved by the *Connector*.

ACO development focused primarily on delivering the second and the third objective. The tools we provided are parts of the Connector, a component that connects a particular argument in the summarized assurance case with its most related information in an external document, such as requirements and source code.

The Connector currently consists of two tools. The first tool, called the Document Connector, connects a list of query keywords with a requirement document in PDF format, and finds sections of the document that are most similar to the keywords provided. The second tool, called the Code Connector, connects a list of query keywords with each keyword set generated by the Document Parser, and finds functions in the provided set of source code files that are most similar to the keywords provided.

These tools are currently implemented in Python. The current version of the ACO requires users to enter keywords of interest. Future versions of the ACO will automatically parse and extract keywords from the assurance case for the query.

Document Connector The Document Connector functionality in the ACO connects a list of query keywords with a requirement document (PDF) uploaded by the user. Given the input query, the Connector will return the most relevant section/subsection in the requirement document. This functionality currently relies on (1) metadata present in the PDF, or (2) the document's Table of Contents. The Connector will automatically extract information from the metadata if metadata is present. When metadata is not present, the Connector will attempt to extract information from the Table of Contents. This information extraction could fail in some rare cases (for example, when information in the Table of Contents follows an unknown format), and in these cases an error is returned. The output of the Document Connector is a list of sections from the source document and their associated similarity scores.

Code Connector The Code Connector functionality in the ACO connects a list of query keywords with source code files. The Connector is currently able to handle scripts written in Python and Ada. Similar to the Document Connector usage, the user is currently expected to upload source code and provide a list of query keywords to the Code Connector. Based on the input query, the Connector will return the most relevant function from the source code files.

The Code Connector operates by extracting information about all of the functions and classes declared in the source code files uploaded. The output of the Code Connector is a list of functions and their associated similarity scores. In general, results of the Code Connector tend to have lower similarity scores than results of the Document Connector; we have attributed this to the fact that source code tends to have more keywords than natural language documents.

3.5 ACCELERATE Exemplar Assurance Cases

During the duration of the ACCELERATE project, our team created two main assurance cases, one for the purpose of developing ACCELERATE, and another for its final assessment. Boeing provided data for an assurance case to be made for the Boeing AH-64D Longbow Apache helicopter. However, because the data are CUI and subject to Boeing licensing terms, they could not be shared with JHU/APL subcontractors. Therefore, the geofence assurance case was created for the purpose of developing ACCELERATE technology, namely its logic, defeaters in the logic, automated assurance argument assembly (A4), templates and argument template library (ATL) and the ACCELERATE argument browser (A2B). Our team successfully delivered a version of ACCELERATE capable of using the A4 and comma-separated values (CSV)-file evidence to instantiate the full geofence assurance case described in this section from templates within the ATL. The case can then be scored by the ACCELERATE logic library and a user can browse the results within the A2B. Our team also delivered a Boeing assurance case in JSON format, which can be uploaded to the ACCELERATE tool, where it can be viewed by users through the A2B and scored with the ACCELERATE logic library.

3.5.1 Geofence Assurance Case

This section describes an assurance case for a cage ceiling monitor (commonly referred to in this document as the "geofence assurance case"). This assurance case is a "correct by construction" case developed by the ACCELERATE team to drive development in preparation for the final assessment of Phase 1 of the ARCOS program. The geofence assurance case is motivated by five monitors defined in [9] for monitoring the movement of a drone and triggering an alarm if and only if it will breach a cage boundary. Five monitors are defined to monitor minimum latitudinal and longitudinal movement, maximum latitudinal and longitudinal movement, and maximum vertical movement. The monitor for vertical movement runs on an FPGA onboard the Intel Aero platform. The geofence assurance case focuses only on the vertical movement monitor.

While the primary focus of the Phase 1 assessment was the Boeing AH-64D system, we found value early on in exploring this additional assurance case for a variety of reasons. The geofence assurance case is not subject to any export controls, and thus can be shared freely with all team members. In addition, the geofence assurance case is small yet rich enough to allow us to refine the ACCELERATE logic in general and to better understand defeater logic in particular. It also allowed us to explore S-Taliro, a plugin to the TA1 Lockheed Martin team's CertGATE tool. As described further in Section 3.5.1.6, S-Taliro found a trajectory that falsifies a metric temporal logic (MTL) formula describing soundness of the vertical cage ceiling monitor.

3.5.1.1 Cage Ceiling Monitor Overview

We give cage ceiling monitors for keeping a drone below a certain altitude. They are inspired by the cage monitors of [9]. Each is a symbolic finite automaton and thus more expressive than Büchi automata used in [9] since they take into account history. They are also more detailed, accounting for the sampling rates of sensors. Sensors approximate vertical and horizontal speeds. The cage ceiling monitor in [9] is augmented with another monitor for monitoring sensors inputs, namely from an onboard electronic barometer BMP (Barometric Pressure sensor), which is used to measure altitude. We shall write *bmp.alt* to denote this altitude. Our monitors are symbolic finite automata with lambda bindings used for capturing history.

The first monitor, the vertical speed monitor, prescribes our expectations for the use of bmp in calculating vertical speed. The other is the cage ceiling monitor, which monitors for proximity to the ceiling and the drone's ability to avoid breaching it based on its vertical speed. The cage ceiling monitor assumes vertical speed is approximated by the distance between two consecutive bmp sensor samples [9]. Since these samples are clocked at a fixed rate, bigger differences between consecutive samples imply greater speed. A sensor may fail to produce consecutive readings at any time and this must be accounted for when measuring vertical speed. Depending on the drone's altitude, missed samples may be tolerated. We take the last known measured vertical speed of the drone to be the drone's actual speed as long as the drone is not nearing the fence as measured by its real altitude. Otherwise, an alarm is generated. This behavior manifests itself automatically when we calculate the tensor product. There is no need to manually create it. More precisely, the tensor product specifies the behavior automatically.

Commercial electronic barometers can be sampled anywhere between 120Hz to 157Hz (see for example Arduino BMP180/280). Depending on the sampling rate, the actual altitude of the drone may be greater than *bmp.alt* . The difference between actual altitude and *bmp.alt* becomes greater as the maximum vertical speed increases and sampling rate decreases. We define the *errormargin* as maximum vertical drone speed divided by the barometer sampling rate. For example, *errormargin* for a drone with a maximum vertical speed of 50ft/sec and with a barometric sensor having sampling rate 25Hz, would be 2ft. This margin must be accounted for when determining if the drone is approaching the ceiling.

Source code is derived from the tensor product of the two monitors. This is the code that runs on an FPGA on the drone to trigger a failsafe operation or recovery function if the drone will breach the ceiling. (See Appendix A for the source code.) This is the code subject to testing and verification with respect to formally-stated requirements.

3.5.1.2 Vertical Speed and Cage Ceiling Monitors

The vertical speed and cage ceiling monitors are shown in Figure 11. They run on a discrete-timed state sequence (σ, τ) where σ is a sequence of states $\sigma_1, \sigma_2, \ldots$ and τ is a monotonically increasing discrete time sequence $\tau_1, \tau_2, \ldots$. The variables defined in a state σ_i are $\sigma_i.bmp$ (barometric pressure), $\sigma_i.cur$ alt (current altitude), $\sigma_i.bmp.alt$ (altitude approximated by $\sigma_i.bmp$) and $\sigma_i.ver$ speed (vertical speed). Note that *samplerate*, *errormargin* and *MAX_ALT* are constants. Variable *t* ranges over the elements of τ. A transition is made on (σ_i, τ_i) if and only if $\sigma_i \models p$ where *p* is the proposition on that transition with all occurrences of *t* replaced by σ_i. We say that a monitor *M* accepts a discrete-timed state sequence (σ, τ) if for every $(\sigma_i, \tau_i) \in (\sigma, \tau)$, there's a transition in *M* on (σ_i, τ_i) when run on the sequence. In the case of acceptance, we write $(\sigma, \tau) \in L(M)$. If there's a (σ_i, τ_i) in the sequence on which *M* cannot make a transition, then we say *M* gets stuck on the sequence.

3.5.1.3 Tensor Product of Monitors

The tensor product of the vertical speed and cage ceiling monitors, vertical speed × *cage ceiling*, is defined in Figure 12 and Figure 13. It has five states and the propositions on transitions are shown on the right.

Notice that in order to satisfy the constraint on $a, 3 \rightarrow b, 2$, the condition highlighted in blue cannot be satisfied when $ver\_speed = y.ver$ speed because it contradicts the necessary condition highlighted in red since $cur\_alt = bmp.alt$. That tells us that we will not rely on the last known vertical

speed when the fidelity of *bmp* is in question and the drone is over *MID_ALT*. This is wise as the drone is nearing the fence, so we demand fidelity in *bmp* or cause an ALARM.

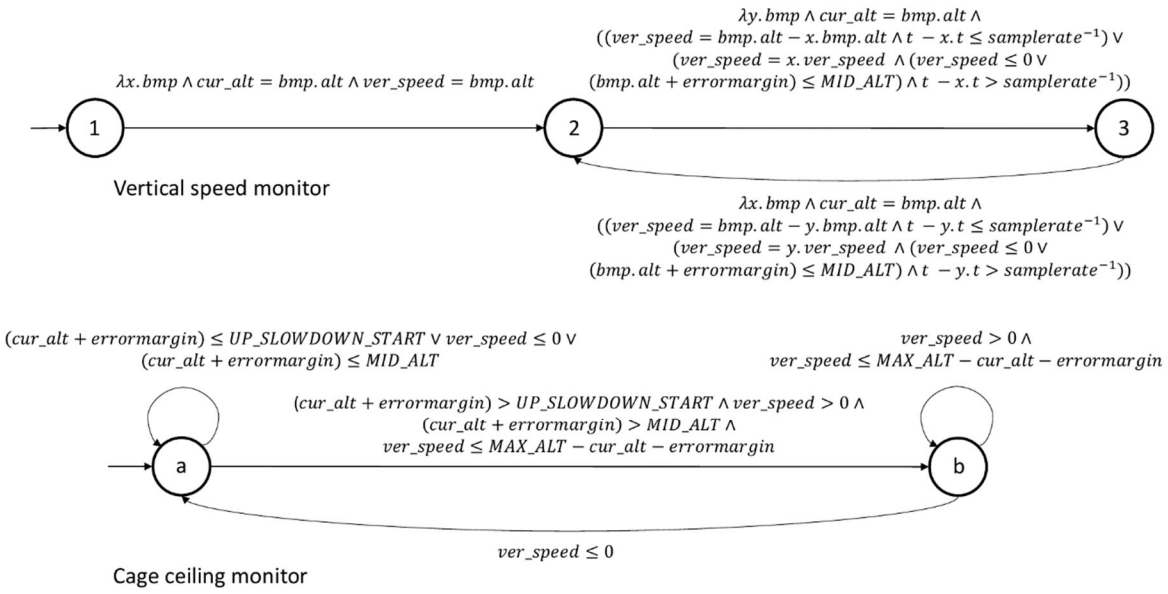


Figure 11. Vertical speed and cage ceiling monitors

For every timed state sequence (σ, τ),

$$(\sigma, \tau) \in L(\text{vertical speed} \times \text{cage ceiling})$$

only if σ models the LTL formula

$$G(\neg in\_slow\_zone \vee \neg nearing\_fence \vee ver\_speed \leq 0 \vee breach\_avoidable)$$

where *in slow zone* is defined as

$$(cur\_alt + errormargin) > UP\_SLOWDOWN\_START$$

nearing fence is defined as

$$ver\_speed > 0 \wedge (cur\_alt + errormargin) > MID\_ALT$$

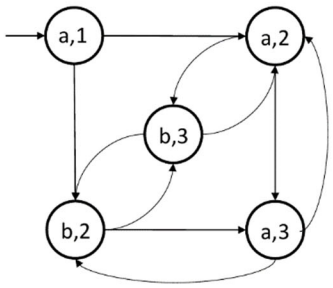
and *breach avoidable* is defined as

$$ver\_speed \leq MAX\_ALT - cur\_alt - errormargin$$

This tells us that the tensor product can monitor a drone's behavior to ensure it always flies freely below the ceiling or toward it while avoiding a breach. Otherwise, it triggers an alarm.

The tensor product is implemented in C by a predicate *within_cage_max_alt*, which returns true if the state sequence it has observed thus far does not violate the linear-time temporal logic (LTL) formula, and false otherwise. In practice, it would be implemented as a background process that is

silent as long as no violation is detected. Otherwise a failsafe operation is triggered. C code is derived from the tensor product and is shown in Appendix A.



a,1 → a,2	$\lambda x..\ bmp \wedge cur\_alt = bmp.alt \wedge ver\_speed = bmp.alt \wedge (cur\_alt + errormargin) \leq UP\_SLOWDOWN\_START \vee ver\_speed \leq 0 \vee (cur\_alt + errormargin) \leq MID\_ALT$
a,1 → b,2	$\lambda x..\ bmp \wedge cur\_alt = bmp.alt \wedge ver\_speed = bmp.alt \wedge \wedge (cur\_alt + errormargin) > UP\_SLOWDOWN\_START \wedge ver\_speed > 0 \wedge (cur\_alt + errormargin) > MID\_ALT \wedge ver\_speed \leq MAX\_ALT - cur\_alt - errormargin$
b,2 → a,3	$\lambda y.\ ver\_speed \leq 0 \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - x.bmp.alt \wedge t\ - x.t \leq samplerate^{-1}) \vee (ver\_speed = x.ver\_speed\ \wedge t\ - x.t > samplerate^{-1}))$
b,2 → b,3	$\lambda y.\ ver\_speed > 0 \wedge ver\_speed \leq MAXALT - cur\_alt - errormargin \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - x.bmp.alt \wedge t\ - x.t \leq samplerate^{-1}) \vee (ver\_speed = x.ver\_speed\ \wedge (bmp.alt + errormargin) \leq MID\_ALT \wedge t\ - x.t > samplerate^{-1}))$
b,3 → b,2	$\lambda x.\ ver\_speed > 0 \wedge ver\_speed \leq MAX\_ALT - cur\_alt - errormargin \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - y.bmp.alt \wedge t\ - y.t \leq samplerate^{-1}) \vee (ver\_speed = y.ver\_speed\ \wedge (bmp.alt + errormargin) \leq MID\_ALT \wedge t\ - y.t > samplerate^{-1}))$

Figure 12. Tensor product with transitions

3.5.1.4 *Within _cage_max_alt* is Sound and Complete for the Tensor Product

Consider calling *within_cage_max_alt* repeatedly on each sample in a sequence of barometric pressure samples $bmp_1, bmp_2, \ldots$. Each sample approximates an altitude, which we write as $bmp_1.alt$, $bmp_2.alt$ and so on. The samples are produced at a fixed rate called the sample rate. The electronic barometer may fail to produce samples on some cycles so the time between each sample may exceed the inverse of the barometer's hardcoded sample rate. This is when the vertical speed monitor demands vertical speed to be the last known speed. Otherwise, the calculated speed can be misleading. As the drone moves above *MID_ALT*, falling back to the last known speed is no longer an option because we demand sensor fidelity. However, the time between two consecutive samples may be sufficiently large such that the calculated vertical speed exceeds

$$MAX\_ALT - cur\_alt - errormargin$$

while, in reality, the drone is moving closer to *MAX ALT* slowly enough such that it avoids breaching the ceiling. This occurrence will cause *within_cage_max_alt* to produce a false alarm. A sequence of samples causing this behavior would falsify a claim that *within_cage_max_alt* never produces an alarm unless the drone will breach the ceiling. Such a sequence defeats the claim. While we can manually prove that *within_cage_max_alt* is sound and complete for the tensor product, unless we make claims and give evidence about how the sensor behaves in operation, we will not be able to relate the behavior of *within_cage_max_alt* to *drone behavior.* Though likely not an issue for electronic barometers, getting measurements in a timely way is an issue for

GPS, since drones can operate in GPS-constrained environments. GPS is used for enforcing horizontal limits on movement within a cage.

Suppose the time at which *within_cage_max_alt* is called is supplied to it via a parameter. Then we have the following statement of soundness for *within_cage_max_alt*:

$$\forall \sigma = \sigma_1, \ldots, \sigma_n.\ \forall \tau = \tau_1, \ldots, \tau_n.$$

$$within\_cage\_max\_alt(\tau_1, \sigma_1.bmp) \wedge \cdots \wedge within\_cage\_max\_alt(\tau_n, \sigma_n.bmp) \Rightarrow$$

$$(\sigma, \tau) \in L(vertical\ speed \times cage\ ceiling)$$

Here the universal quantification is over all state sequences σ that bind at least those variables required by the product monitor (*within_cage_max_alt* needs only *bmp* from a given state).

a,2 → a,3	$\lambda y.((cur\_alt + errormargin) \leq UP\_SLOWDOWN\_START \vee ver\_speed \leq 0 \vee (cur\_alt + errormargin) \leq MID\_ALT) \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - x.bmp.alt \wedge t - x.t \leq samplerate^{-1}) \vee (ver\_speed = x.ver\_speed \wedge (ver\_speed \leq 0 \vee (bmp.alt + errormargin) \leq MID\_ALT) \wedge t - x.t > samplerate^{-1}))$
a,3 → a,2	$\lambda x.((cur\_alt + errormargin) \leq UP\_SLOWDOWN\_START \vee ver\_speed \leq 0 \vee (cur\_alt + errormargin) \leq MID\_ALT) \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - y.bmp.alt \wedge t - y.t \leq samplerate^{-1}) \vee (ver\_speed = y.ver\_speed \wedge (ver\_speed \leq 0 \vee (bmp.alt + errormargin) \leq MID\_ALT) \wedge t - y.t > samplerate^{-1}))$
a,3 → b,2	$\lambda x.(cur\_alt + errormargin) > UP\_SLOWDOWN\_START \wedge ver\_speed > 0 \wedge (cur\_alt + errormargin) > MID\_ALT \wedge ver\_speed \leq MAX\_ALT - cur\_alt - errormargin \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - y.bmp.alt \wedge t - y.t \leq samplerate^{-1}) \vee (ver\_speed = y.ver\_speed \wedge (ver\_speed \leq 0 \vee (bmp.alt + errormargin) \leq MID\_ALT) \wedge t - y.t > samplerate^{-1}))$

a,2 → b,3	$\lambda y.(cur\_alt + errormargin) > UP\_SLOWDOWN\_START \wedge ver\_speed > 0 \wedge (cur\_alt + errormargin) > MID\_ALT \wedge ver\_speed \leq MAX\_ALT - cur\_alt - errormargin \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - x.bmp.alt \wedge t - x.t \leq samplerate^{-1}) \vee (ver\_speed = x.ver\_speed \wedge (ver\_speed \leq 0 \vee (bmp.alt + errormargin) \leq MID\_ALT) \wedge t - x.t > samplerate^{-1}))$
b,3 → a,2	$\lambda x.ver\_speed \leq 0 \wedge bmp \wedge cur\_alt = bmp.alt \wedge ((ver\_speed = bmp.alt - y.bmp.alt \wedge t - y.t \leq samplerate^{-1}) \vee (ver\_speed = y.ver\_speed \wedge t - y.t > samplerate^{-1}))$

Figure 13. Tensor product transitions continued

The converse of the statement is completeness of *within_cage_max_alt* . For completeness to hold, time τ_i at which *within_cage_max_alt* is called must satisfy

$$\tau_i - \tau_{i-1} \leq cycletime$$

Therefore, *within_cage_max_*alt must run to completion within *cycletime*, defined as the inverse of the sampling rate. So we add a claim:

Claim: (Process). *Within_cage_max_alt* runs to completion exclusive of system overhead and without interrupts within the barometer's cycle time.

Soundness and completeness share another subclaim addressing the accuracy and reliability of the electronic barometer. After all, neither soundness nor completeness matter in practice if the altitude

measurements on which they are based and the definition of error margin do not match reality. Thus, we make another claim and assumption:

Claim: (Reality check). The barometer's error in approximating altitude does not exceed the margin of error defined by maximum drone speed divided by the barometer's sample rate. The hard-coded sample rate is observed for the barometer in practice.

Assumption: The electronic barometer is calibrated with the atmospheric pressure at ground level on the day of use, or the atmospheric pressure is obtained in real time through a radio link transmitting ground-level pressure to the barometer during flight.

3.5.1.5 Assurance Case – Putting it all Together

Claim: Five monitors fence vertical, latitudinal and longitudinal drone movement, triggering an alarm whenever the fence will be breached. Drone has maximum speed 100ft/sec and an onboard Arduino BMP280 barometric pressure chip capable of being sampled at a rate of 25Hz and a height measurement range of +27,000ft to -1500ft.

Strategy: Iterate over 5 monitors individually
Justification: Only need to fence min/max lat, min/max long, and max alt.
Assumption: Fencing capability along each axis can be assessed independently.
Claim.1: min lat monitor triggers alarm iff drone is about to breach min latitude.
Claim.2: max lat monitor triggers alarm iff drone is about to breach max latitude.
Claim.3: min long monitor triggers alarm iff drone is about to breach min longitude.
Claim.4: max long monitor triggers alarm iff drone is about to breach max longitude.

Claim.5: *within_cage_max_alt* triggers an alarm if and only if drone equipped with Arduino BMP280 barometric pressure chip will breach ceiling.

Strategy.5: (Divide and conquer). Formally construct tensor product specification of two simple monitors, one monitoring vertical speed, the other potential for ceiling breach. Generate code automatically for *within_cage_max_alt* from the tensor product and show completeness and soundness of code with respect to tensor product specification.

Context.5: Vertical speed, cage ceiling and tensor product monitors given.

Claim.5.1: (completeness). If the tensor product does not get stuck then *within_cage_max_alt* does not trigger an alarm:

$$\forall \sigma = \sigma_1, \ldots, \sigma_n.\ \forall \tau = \tau_1, \ldots, \tau_n.$$

$$(\sigma, \tau) \in L(\textit{vertical speed} \times \textit{cage ceiling}) \Rightarrow$$

$$\textit{within\_cage\_max\_alt}(\tau_1, \sigma_1.\textit{bmp}) \wedge \cdots \wedge \textit{within\_cage\_max\_alt}(\tau_n, \sigma_n.\textit{bmp})$$

Evidence 5.1.1: Formal proof.

Claim 5.1.2: (Process). *within_cage_max_alt* runs to completion within barometer's cycle time.

Evidence 5.1.2.1: Observed running time of *within_cage_max_alt* exclusive of system overhead and without interrupts to be 8msec while the barometer's cycle time is 1/25Hz or 40msec.

Claim 5.2: If drone will not breach ceiling then *within_cage_max_*alt does not trigger an alarm (lift Claim 5.1 to Claim 5 "only-if").

Defeater 5.2.1: the time between two samples may be enough to let the drone move closer to *MAX ALT* slowly enough to avoid breaching the ceiling, yet the calculated vertical speed can exceed $MAX\,ALT - cur\,alt - errormargin$ if the barometer fails to produce samples on consecutive cycles. This will cause *within_cage_max_alt* to trigger an alarm.

Defeater.5.2.1.1: The calculated vertical speed cannot exceed $MAX\,ALT - cur\_alt - errormargin$ unless drone will breach ceiling.

Claim.5.2.1.1.1: (Reality check). The barometer's error in approximating altitude does not exceed the margin of error defined by maximum drone speed divided by the barometer's sample rate. The hardcoded sample rate is observed in practice.

Evidence.5.2.1.1.1.1: Observed relative accuracy of ± 0.12mbar in barometric pressure for BMP280 between 15 - 20 degrees C, or a relative accuracy of ±3ft. Drone's maximum speed is 100ft/sec and the BMP280 is sampled at a rate of 25Hz. So the margin of error is 100/25 = 4ft per cycle. Observed BMP280 can be sampled at rate of 25Hz through testing at sea level (1013.25mb) and air temperature 15 - 20 degrees C.

Table 1. Falsifying trajectory found by S-Taliro assuming 10% sensor failure rate

altitude	*vertical speed*	*within cage max alt*
3.8761	3.8123	1.0000
7.4761	3.6001	1.0000
10.8518	3.3757	1.0000
13.9910	3.1392	1.0000
16.8815	2.8905	1.0000
19.5113	2.6298	1.0000
21.8682	2.3569	1.0000
23.9400	2.0718	1.0000
25.7146	1.7746	0

Assumption.5.2.1.1.2: The electronic barometer is calibrated with the atmospheric pressure at ground level on the day of use, or is obtained in real time through a radio link transmitting ground-level pressure to the barometer during flight.

Claim.5.3: (soundness). If *within_ cage_ max_ alt* does not trigger an alarm then the tensor product will not get stuck:

$$\forall \sigma = \sigma_1, \ldots, \sigma_n.\ \forall \tau = \tau_1, \ldots, \tau_n.$$

$$within\_cage\_max\_alt(\tau_1, \sigma_1.bmp) \wedge \cdots \wedge within\_cage\_max\_alt(\tau_n, \sigma_n.bmp) \Rightarrow$$

$$(\sigma, \tau) \in L(vertical\ speed \times cage\ ceiling)$$

Evidence.5.3.1: Formal proof.

Claim.5.4: (lift Claim.5.3 to Claim.5 "if"). If *within_ cage_ max_ alt* does not trigger an alarm then drone will not breach ceiling.

Assumption.5.4.1: Drone operates in a controlled operating environment.

Defeater.5.4.1.1: Cannot control drone's operating environment or any external conditions affecting its flight.

3.5.1.6 Geofence Assurance Case Collaborations

The JHU/APL team collaborated with Lockheed Martin, a TA1 performer, to produce evidence for the geofence assurance case. Lockheed Martin's subcontractor, Arizona State University, produced a signal temporal logic falsifier tool called S-Taliro [(Annpureddy, Liu, Fainekos, & Sankaranarayanan, 2011)]. The tool was used to find a defeater of the claim "*within_cage_max_alt* cannot yield a false alarm". Indeed, it can, if the barometric pressure sensor fails to produce samples at the rate it expects.

The C code for *within_cage_max_alt* was translated into Matlab for simulation. Assuming the barometric pressure sensor fails to produce sample altitudes 10% of the time for whatever reason, S-Taliro found a trajectory for *within_cage_max_alt* that falsifies the following LTL formula:

$$eqone \vee (in\_slow\ zone \wedge \neg vs\_leq\_zero \wedge \neg breach\ avoidable)$$

It says if *within_cage_max_alt* triggers an alarm (*eqone* is false) then the ceiling will be breached, that is, the drone is in the slow zone and moving at a positive vertical speed that makes a breach unavoidable. The trajectory found for *within_cage_max_alt* is shown in *Table 1*.

In the simulation, the maximum altitude is 30 meters and the error margin is 0.4 meters. In the last triple, you find *within_ cage_ max_ alt* produces false (0) meaning it believes the drone will breach the ceiling. However,

$$in\_slow\_zone \wedge \neg vs\_leq\_zero \wedge \neg breach\_avoidable$$

is false; *in_slow_zone* is defined as over 24 meters, and the drone is at 25.7146 meters, vertical speed is 1.7746 meters/sec, which is positive. However, *breach_avoidable* is true because it is defined as $1.7746 \leq 30 - 25.7146 - 0.4 = 3.8854$. So S-Taliro discovered *within_cage_max_alt* an produce a false alarm if it does not see altitude samples at the rate it expects.

4.0 RESULTS AND DISCUSSION

The artifacts and curated evidence provided for the assessments of the ACCELERATE tool (based on the Boeing AH-64D Apache system) presented a few challenges for our team, resulting in sub-optimal progress in development and a shift of focus.

The assessment artifacts and evidence provided were primarily export-controlled information that was also covered by a Boeing Software License Agreement. This greatly limited sharing of information both internal to our team, and with other performers on the project. Due to limited shareability, we focused our team on the geofence assurance case presented in this document. This served as an excellent example to explore and build out our capabilities. However, it was not presented to the evaluation team due to the assessment's focus on the Apache system. We shifted resources during the ACCELERATE program to better meet these data requirements; however, we maintain the position that an open source example would aid in faster progress and fuller collaboration amongst performers.

During the creation of our Boeing assurance case, our progress was also limited due to gaps in the evidence and artifacts provided, and in our ability to programmatically discover and incorporate requirements and evidence data from RACK.

Our attempts to apply clustering approaches to support automatic assurance case instantiation from templates also faced challenges due to the lack of data (e.g., artifacts, evidence, defeaters, etc.).

5.0 CONCLUSIONS & FUTURE WORK

This section provides an overview of future avenues of research for the ACCELERATE project and expansion of the current capabilities within each of the modules of ACCELERATE.

5.1 Logic Library

The ACCELERATE Logic Library has served well for the estimation of assurance and confidence in the examples that we explored during the development of the ACCELERATE tool. As future work, we propose expanding the breadth of use cases to test the ACCELERATE logic, exploring more types of defeaters and security arguments. In addition, we would like to develop methods to enable conversion of more types of evidence to Beta PDFs for scoring. This would benefit from collaboration with evidence providers (e.g., TA1 performers) to determine appropriate scores for the evidence produced by their tools. We are also refining our logic library to provide error bars on the resulting score. This will aid certifiers in determining the effectiveness of a given assurance case.

5.2 Argument Template Library

Argument Templates Now that we have a baseline set of argument templates in the ATL, we want to iterate on more examples to determine whether generic templates (such as "Requirements Decomposition", etc.) are universally useful to assurance argument instantiation, or if more specific templates are necessary to produce a meaningful argument. During this process, we will also determine whether the set of generic templates currently in the ATL is complete, and if not, which additional templates are needed to enable instantiation of a sufficient range of assurance arguments.

Generation of New Argument Templates with NLP We developed an initial capability for the generation of new argument templates from NLP of source documents. We would like to develop additional capabilities to further test the effectiveness of this approach. If general templates are deemed inadequate for the automatic instantiation of a new assurance arguments, NLP generation of more specific argument templates could be a promising alternative. We also plan to explore which input document(s) produce the best NLP-generated templates for ACCELERATE (e.g., requirements documents, certification documents, or a combination of inputs).

5.3 Automatic Assurance Argument Assembler

We would like to continue our work towards automated construction of assurance arguments with the A4 module. Our focus would be on automating the composition of templates and instantiating this template composition with data from the RACK to form a completed high-quality case, which can then be scored by the logic. In addition, we expect further integration with our Logic work, particularly in the identification and representation of defeaters.

5.3.1 Machine Learning-supported A4

After de-emphasizing clustering due to limitations in training data (e.g., evidence, argument templates, etc.), we plan to explore using machine learning (ML) algorithms to optimize assurance case instantiation. An optimized A4 would guarantee that the best argument case is selected for a

given claim. It would also ensure that the assurance case (AC) is complete for a given system, based on the evidence and artifacts available, and the defeaters applied.

Assurance Case Evaluation Metrics Before evaluating the best ML approach(es) for optimizing AC instantiation, we would define metrics representing the quality of an assurance case. This is an open research problem that we plan to target by combining and quantizing results from different ACCELERATE modules including the logic library and the ATL.

We will use Siemens' "artifact coverage" and "system property coverage" metrics to quantify the amount of evidence and artifacts used during an assurance case instantiation. Then, we will instantiate assurance cases of different level of coverage, with different combinations of defeaters, and score them with the logic library. We will use the collection of resultant confidence scores from the logic library, along with the corresponding assurance cases metadata, to feed an ML model. We expect the ML model to provide the combination of defeaters that makes the assurance case strongest given the set of input evidence and artifacts. This ML model may also provide a mechanism to incorporate certifier feedback on the clarity of produced assurance cases; in order to drive the A4 to produce high-quality assurance cases that are also comprehensible.

Best ML Approach As template availability was one of the challenges faced by the A4 team, future work may aim for semi-supervised or Bayesian machine learning algorithms that perform better for small datasets. In future work, the A4 could also move away from the use of argument templates to train the ML models, and utilize raw evidence and artifacts from the RACK for this purpose. Even if templates are limited, they contain a larger set of embedded data that can still be combined with ML to build a high-quality assurance case. This new method favors a more general, non-deterministic, and fully-automated way of building an assurance case over the use of templates with pre-defined parent-child relationships.

5.4 Argument Comprehensibility Optimizer

As a repository of assurance cases are produced by the other modules of ACCELERATE and other ARCOS teams, the ACO proposed effort may be further developed to identify and explore leveraging recurring idioms in assurance cases.

Future efforts in ACO development may also include the development of methods in Bayesian inference to reveal recurring idioms in assurance cases. These idioms can then be leveraged to display the assurance argument to the user in a meaningful way with varying levels of granularity, allowing the user to adjust the display of the assurance argument according to their preferences, and to dive deeper into specific sub-argument(s) of interest.

5.5 Data Ingest

The NLP-based argument template generation features of the ACCELERATE tool could also be leveraged to translate system artifacts into a machine-readable format that could be ingested into the ARCOS curated evidence store. Further development would be required to translate these features into a general tool capable of ingesting evidence into the RACK curated evidence store; but we believe this work would benefit the ARCOS program as a whole as it would greatly improve

the availability and granularity of system documents and information available to TA3 argument generation teams.

5.6 User Interface

The ACCELERATE user interface developed during this effort is very much a development prototype. Future work should include a comprehensive usability assessment, or replacement of this user interface with a more mature toolset. In addition, future work should incorporate the ACO module into the frontend of ACCELERATE, in addition to the NLP-based generation of new argument templates.

5.7 External Activities and Transition

To further advance and broaden the impact of our work, we have worked to report out to the broader research community and transition our tools and technologies to other government organizations. We are also working to get open source approval for the ACCELERATE software.

Publications/presentations include "Lifting Formal Proof to Practice via an Assurance Case" in HCSS 2022 (Brule, Helble, Thober, & Volpano, May 2022) and "Automating Pattern Selection for Assurance Case Development of Cyber- Physical Systems", to appear in SAFECOMP 2022 (Ramakrishna, Jin, Dubey, & Ramamurthy, September 2022). We also have a draft document on our logic formalism "A probabilistic logic of Beta distributions and efficient, approximate Bayesian inference", which we intend to post to arXiv [(arXiv., n.d.)] in the near future.

We have also successfully transitioned our ACCELERATE tools and technologies to use on projects for other government organizations. Specifically, we have been using ACCELERATE to explore building assurance cases in support of efforts funded by the DOE Office of Cybersecurity, Energy Security, and Emergency Response (CESER), and are working on using and improving ACCELERATE as part of the Sentinel program with the Air Force Nuclear Weapons Center (AFNWC).

5.8 Final Conclusion

The TA3 ACCELERATE team led by JHU/APL produced an initial version of the ACCELERATE tool, capable of automatic construction and assessment of an assurance case argument. ACCELERATE consists of five modules, the first four driving at a different innovative claim: (1) ACCELERATE Logic Library - *Rigorous* definition of a novel variant of defeasible logic for representing assurance cases, (2) ATL - NLP techniques to abstract certification documents into a library of Expressive argument templates, (3) A4 - advances to clustering, graph-matching, and transfer learning algorithms to ensure Efficient instantiation of argument templates with evidence, and (4) Argument Comprehensibility Optimizer (ACO) - improvements to interpretable machine learning to reveal Comprehensible associations between evidence and arguments, and the fifth module, the ACCELERATE Argument Browser (A2B), serving as a user interface into the system.

We delivered ACCELERATE v1.2.1, comprising implementations of each of these modules, along with two assurance cases: one for a geofence cage ceiling monitor, a compelling case for exploration of defeaters and development of ACCELERATE, which can be automatically instantiated

within the ACCELERATE tool, viewed by users, and scored with the ACCELERATE logic library; and an initial version of an assurance case for the Boeing AH-64D Apache Longbow Attack Helicopter, which can be viewed and scored within the ACCELERATE framework.

As a potential future work, we propose advancing the development of the ACCELERATE with a focus on exploration of more types of defeaters and security arguments in the logic library, refinement of NLP generation of argument templates with the ATL, further automation of assurance argument construction and discovery of defeaters in the A4, and assurance argument abstraction in the ACO. In addition, we propose continuing the development of the frontend of ACCELERATE, the A2B, and combining our work in NLP processing to the task of data ingestion to improve automation.

6.0 REFERENCES

[1] Annpureddy, Y., Liu, C., Fainekos, G., & Sankaranarayanan, S. (2011). S-Talior: A Tool for Temporal Logic Falsification for Hybrid systems. *International Conference onTools and Algorithms for the Construction and Analysis of Systems (TACAS).*

[2] *arXiv.* (n.d.). Retrieved from https://arxiv.org

[3] Atighetchi, M. (2009). *XDDS Installation, Administration, and Demonstration Guide.* User's Guide, BBN Technologies, Cambridge.

[4] Brule, J., Helble, S., Thober, M., & Volpano, D. (May 2022). Lifting formal Proof to Practice via an Assurance Case. *The 22nd annual High Confidence Software and Systems Conference (HCSS 2022).*

[5] Goodhart, C. A. (n.d.). *Problems of monetary management: the UK experience* (1984 ed., Vol. Monetary Theory and Practice). Springer.

[6] Jaynes, E. T. (2003). Probability theory: The logic of science. *Cambridge University Press*.

[7] The Johns Hopkins University Applied Physics Laboratory. (January 6, 2022). *ARCOS ACCELERATE Phase 1 Report.*

[8] Lamport et al, L. (2018). If You're Not Writing a Program, Don't Use a Programming Language. *Bulletin of EATCS 2.125*.

[9] Ramakrishna, S., Jin, H., Dubey, A., & Ramamurthy, A. (September 2022). Automating Pattern Selection for Assurance Case Development of Cyber-Physical Systems. *The 41st International Conference on Computer Safey,Reliability and Security (SAFECOMP).*

[10] Stamenkovich, J., Maalolan, L., & Patterson, C. (2019). Formal Assurances for Autonomous Systems without Verifying Application Software. *Workshop on Research, Education and Development of Unmanned Aerial Systems*, 60-69.

APPENDIX A: DERIVED C CODE FOR GEOFENCE CAGE CEILING MONITOR

```
#include <stdint.h>#include <inttypes.h>
#include <stdbool.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define SAMPLINGRATE 1    /* Hz */
#define MAXSPEED 2    /* > 0 meters per sec */
#define MID_ALT  20    /* meters */
#define MAX_ALT  30    /* meters */
#define UP_SLOWDOWN_START 24   /* meters */

#define breach_avoidable(X, Y) ((X) <= (MAX_ALT - (Y) - errormargin))
#define nearing_fence(X, Y) (((X) > 0) && (((Y) + errormargin) > MID_ALT))

typedef struct {
 int32_t alt;
} bmp;

const uint32_t errormargin = 1 + ((MAXSPEED - 1)/SAMPLINGRATE);
const double cycletime = 1/(double)SAMPLINGRATE;



bool within_cage_max_alt(const bmp *b)
{

enum sfaStates {a1, a2, a3, b2, b3, ALARM};
static enum sfaStates state = a1;
static int32_t alt = 0;
static int32_t speed;
static time_t time1, time2;

int32_t ver_speed = b->alt - alt; /* negative on descent */
```

```
bool in_slow_zone = ((b->alt + errormargin) > UP_SLOWDOWN_START);

time(&time2);    /* current time */

switch (state) {
 case a1:
  if (!in_slow_zone || !nearing_fence(ver_speed, b->alt))
   state = a2;
  else
   if (breach_avoidable(ver_speed, b->alt))
    state = b2;
   else state = ALARM;
  break;

 case a2:
  if (!in_slow_zone || !nearing_fence(ver_speed, b->alt))
   if (difftime(time2, time1) < cycletime)
    state = a3;
   else {
    /* speed is defined because a2 is unreachable in initial call
    * of within_cage_max_alt and speed is defined after initial call
    */
    ver_speed = speed; /* redefine as last known vertical speed */

    if (!nearing_fence(ver_speed, b->alt))
     state = a3;
    else state = ALARM;
   }
  else
   if (breach_avoidable(ver_speed, b->alt)) {
    if (difftime(time2, time1) < cycletime)
     state = b3;
    else state = ALARM;
   }
   else state = ALARM;
  break;
```

```
case a3:
 if (!in_slow_zone || !nearing_fence(ver_speed, b->alt))
  if (difftime(time2, time1) < cycletime)
   state = a2;
  else {
   ver_speed = speed;
   if (!nearing_fence(ver_speed, b->alt))
    state = a2;
   else state = ALARM;
  }
 else
  if (breach_avoidable(ver_speed, b->alt))
   if (difftime(time2, time1) < cycletime)
    state = b2;
   else state = ALARM;
  else state = ALARM;
 break;

case b2:
 if (ver_speed <= 0)
  if (difftime(time2, time1) < cycletime)
   state = a3;
  else {
   ver_speed = speed;
   if (ver_speed <= 0)
    state = a3;
   else
    if (breach_avoidable(ver_speed, b->alt))
     if (!nearing_fence(ver_speed, b->alt))
      state = b3;
     else state = ALARM;
    else state = ALARM;
  }
 else
  if (breach_avoidable(ver_speed, b->alt))
```

```
   if (difftime(time2, time1) < cycletime)
    state = b3;
   else {
    ver_speed = speed;
    if (ver_speed <= 0)
     state = a3;
    else
     if (breach_avoidable(ver_speed, b->alt))
      if (!nearing_fence(ver_speed, b->alt))
       state = b3;
      else state = ALARM;
     else state = ALARM;
   }
  else state = ALARM;
 break;

case b3:
 if (ver_speed <= 0)
  if (difftime(time2, time1) < cycletime)
   state = a2;
  else {
   ver_speed = speed;
   if (ver_speed <= 0)
    state = a2;
   else
    if (breach_avoidable(ver_speed, b->alt))
     if (!nearing_fence(ver_speed, b->alt))
      state = b2;
     else state = ALARM;
    else state = ALARM;
  }
 else
  if (breach_avoidable(ver_speed, b->alt))
   if (difftime(time2, time1) < cycletime)
    state = b2;
   else {
```

```
     ver_speed = speed;
     if (ver_speed <= 0)
      state = a2;
     else
      if (breach_avoidable(ver_speed, b->alt))
       if (!nearing_fence(ver_speed, b->alt))
        state = b2;
       else state = ALARM;
      else state = ALARM;
    }
   else state = ALARM;

  break;
 default:
  state = ALARM;
 }

time1 = time2;
alt = b->alt;
speed = ver_speed;

return (state != ALARM);
}
```

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

A2B	ACCELERATE Argument Browser
A4	Automated Assurance Argument Assurance Cases
AC	Assurance Case
ACCELERATE	Assurance Cases for Certification of Efficiently Learned Evaluated Rapidly and Automatically using Theory of Evidence
ACO	Argument Comprehensibility Optimizer
AFNWC	Air Force Nuclear Weapons Center
API	Application Programming Interface
APL	Applied Physics Laboratory
ARCOS	Automated Rapid Certification of Software
ATL	Argument template library
BMP	Barometric Pressure
CESER	Cybersecurity, Energy Security, and Emergency Response
CSV	Comma Separated Values
CUI	Controlled Unclassified Information
DOD	Department of Defense
DOE	Department of Energy
FPGA	Field Programmable Gate Array
GPS	Global Positioning System
GSN	Goal Structured Notation
HCSS	High Confidence Software and Systems
JHU	Johns Hopkins University
JSON	JavaScript Object Notation
LLC	Limited Liability Company
LTL	linear-time temporal logic
ML	Machine Learning
MTL	Metric Temporal Logic
NLP	Natural Language Processing
PDF	Probability Density Function
RACK	Rapid Assurance Curation Kit
REST	Representational State Transfer
TA	Technical Area