Continuous safety & security evidence generation, curation and assurance case construction using the Evidential Tool Bus

Natarajan Shankar, Minyoung Kim, Huascar Sanchez
SRI International, Menlo Park, CA, USA
Email: {shankar, mkim, hsanchez}@csl.sri.com

Harald Rueß, Tewodros Beyene, Radouane Bouchekir
fortiss GmbH, Munich, Germany
Email: {ruess, beyene, bouchekir}@fortiss.org

Devesh Bhatt, Srivatsan Varadarajan, Anitha Murugesan, Hao Ren, Isaac Hong-Wong
Honeywell Research, Plymouth, MN, USA
Email: {devesh.bhatt, srivatsan.varadarajan, anitha.murugesan, hao.ren2, isaachong.wong}@honeywell.com

Kit Siu, Sarat Chandra Varanasi
GE Aerospace Research, Niskayuna, NY, USA
Email: {siu, saratchandra.varanasi}@ge.com

Michael D. Ernst
University of Washington, Seattle, WA, USA
Email: mernst@cs.washington.edu

***Abstract*—Establishing assurance of software is indispensable in safety-critical systems. Constructing an assurance case for the safety & security of software subsumes the entire development and V&V workflow involving the use of a multitude of (formal) analysis tools to develop claims supported by diverse sets of evidence. This evidence needs to be curated for certification and assurance case construction. Further, the complexity of information flows gets compounded due to changing needs & goals over the course of certification. We demonstrate the application of the Evidential Tool Bus (ETB2), on an industrial use case workflow involving several tools and methodologies, to support continuous Evidence Generation, their Curation & Assurance Case Construction, from major industrial collaborators in the aviation industry. Evidence Generation follows the Design for Certification (DesCert) methodology. Curation uses the Rapid Assurance Curation Kit (RACK) for semantic reification of evidential data. Assurance case tools then use the curated evidence from RACK for assurance case construction.**

***Keywords*—Continuous Assurance, Certification Evidence Generation, Certification Evidence Curation**

1. Introduction

Establishing assurance of safety-critical systems is becoming increasing complex due to the increasing complexity of missions, architecture, and pervasive use of software. Constructing an assurance case argument for the safety and security of systems subsumes the entirety of development and V&V processes ranging from requirements capture, design iterations, code construction for satisfying certification objectives prescribed by industry standards such as DO-178C/ARP4754A. This involves the construction of multiple development/verification artifacts using generation and formal analysis tools — creating sets of claims supported by evidence. Further, the complexity of information flows gets compounded due to changing requirements, the different tools used, and evolving processes that occur over the course of certification.

In this paper, we demonstrate workflow automation for an industrial use case, involving several tools and methodologies for Certification Evidence Generation and, their Curation and Assurance Case Construction, from major industrial collaborators in the aviation industry. The Certification Evidence Generation workflow in our case study follows the Design for Certification (DesCert) methodology [1] centered around three main thrusts – application of mathematical rigor through the use of formal models and modeling languages to establish safety and security properties of software, efficiency of arguments in terms of the ease of identifying their defeaters, and composability of abstractions for a system of systems. The Constrained Language Enhanced Approach to Requirements (CLEAR) modeling language is used to capture the ontology associated with the requirements, architecture, assurance claims, and the software and evidential artifacts, along with provenance data.

The evidence generation workflow involves modeling various aspects of the system at the levels of system architecture, requirements, and code. Several analysis and V&V tools are used on the system modeling artifacts to verify various properties necessary to establish safety and security assurance objectives.

The evidence curation workflow involves using the Rapid Assurance Curation Kit (RACK) ontology and tools. RACK provides data ingestion to facilitate the triplification of certification evidence into a semantic triple-store, and perform a wide variety of ontology and evidential data checks to ascertain the quality of evidence.

The Evidential Tool Bus (ETB2), is used to capture formally-defined workflows and orchestrate the respective tool executions in a completely distributed fashion. Claims about tool executions and the results obtained from their runs are registered when an ETB2 workflow is triggered, and they are continuously maintained upon changes to evidence, the tools used, or the workflow itself. ETB2 thus facilitates continuous maintenance of evidence and assurance case arguments with continuously changing requirements and design decisions.

Upon curating evidence to sufficient quality, assurance case tools can query RACK to utilize the evidence for appropriate assurance case construction. The assurance cases may be authored in a format that follows a certain methodology, such as using Goal-structuring Notation (GSN) or the more novel Claims-Arguments-Evidence (CAE) framework.

We perform a case study integrating these tools and methodologies applied to a safety and security case of an Arducopter platform containing a Proxy Ground Controller, Runtime Monitor/Logger, and an Advanced Fail Safe running inside a Secure Robot Operating System (SROS 2) instance. With changing requirements, ETB2 orchestrates an integrated pipeline to invalidate safety/security claims that lead back through to evidence ingested into RACK and ultimately back to the sources of evidence provided by DesCert tools.

Related Work While DevOps is a widely accepted process towards continuous software development and integration, the application of such methodologies towards continuous assurance is novel. Hubbs *et al.* [2] note the importance of a robust pipeline for safety-critical software development, considering objectives in standards such as DO-178C. They apply their "Cert Dev Ops" methodology on a small scale flight management application, claiming a 50% certification-related documentation reduction. Ribiero *et al.* have studied the applicability of Agile Methodologies and mention "Cert Dev Ops" for Aerospace safety-critical software development. de La Vara *et al.* [3] study the need for broader tool support for evidence management for assurance of safety-critical systems. Glerischer *et al.* [4] propose a framework formalizing and performing step-wise refinement of assurance cases with changing evidence and apply it on an autonomous robot case study. However, tool support and workflow specification is not formalized in the above work. Bensalem *et al.* [5] have used ETB for continuous assurance of learning-enabled autonomous systems for machine learning lifecycle activities. Sorokin *et al.* [6] have applied ETB for continuous assurance for a case study from the automotive domain. However, these works do not address formal ontology-based exchange among tools of system modeling, properties, and other evidence, along with workflow provenance.

2. DesCert Methodology and Workflow Automation

Figure 1 shows the high-level workflow of development and verification activities, subsequent evidence curation, and construction of assurance case for safety and security. We introduce below the workflow ontology and then each part of the workflow:

DesCert Workflow Ontology A comprehensive ontology captures all aspects of the workflow including provenance and relationships of all the *evidence* – comprising of activities, entities (artifacts produced), and agents (e.g., tools), declared as instances of specific ontology classes. This ontology-first approach in development aims to formally describe the creation and consumption artifacts by various development/verification activities with complete provenance and traceability – enabling construction of assurance cases.

The DesCert ontology is expressed in the CLEAR notation and includes specific classes and corresponding object instances for all the activities, artifacts, and tools shown in Figure 1. An integral part of our workflow automation approach is that each activity invocation, using a particular tool, creates all the related provenance ontology data (which include references to actual artifacts) and passes it along to other activities. Each activity uses the ontology data pertinent to it and retrieves referenced artifacts. All ontology data and artifacts flow to final compilation of all the evidence. A comprehensive report on the application of DesCert methodology for evidence generation on the ArduCopter AFS is detailed elsewhere [1].

Property-Based Approach A central aspect of our approach is property-based assurance [7]; i.e., assurance is achieved by specifying properties about various aspects of the system (covering safety and security) and then providing evidence that those properties are satisfied. Examples of some of the properties are shown at the bottom of Figure 1. Some of the property results are created by construction, e.g., using underlying architecture/OS guarantees; other property results are created using analysis tools on the system development artifacts. All properties and results are specified using CLEAR notation and contain reference to the scope (system element) of each property and evidence of property results. The provenance ontology data for properties and evidence is also specified using CLEAR; all data then flow to the evidence compilation.

System Modeling Activities The development starts with *System Modeling*, which includes the creation of system ConOps, architecture, system requirements, software high-level requirements, code components. Additionally, a security model is created specifying attacks, vulnerabilities, and mitigation by properties/controls, each referring to architecture elements (touch points). System modeling is primarily a manual effort supported by the CLEAR notation/IDE. Properties are also specified as part of modeling each pointing to a specific scope within system architecture. This allows the result of each property to be created by downstream analysis/verification activities.

Analysis and Verification Activities The assurance workflow includes several analysis and V&V activities, shown at the bottom of Figure 1, following our property-based assurance approach. Each activity uses specific tools to prove a subset of properties relevant to it and creates property results along with relevant provenance ontology data. Any relevant artifacts from system modeling are used as inputs.

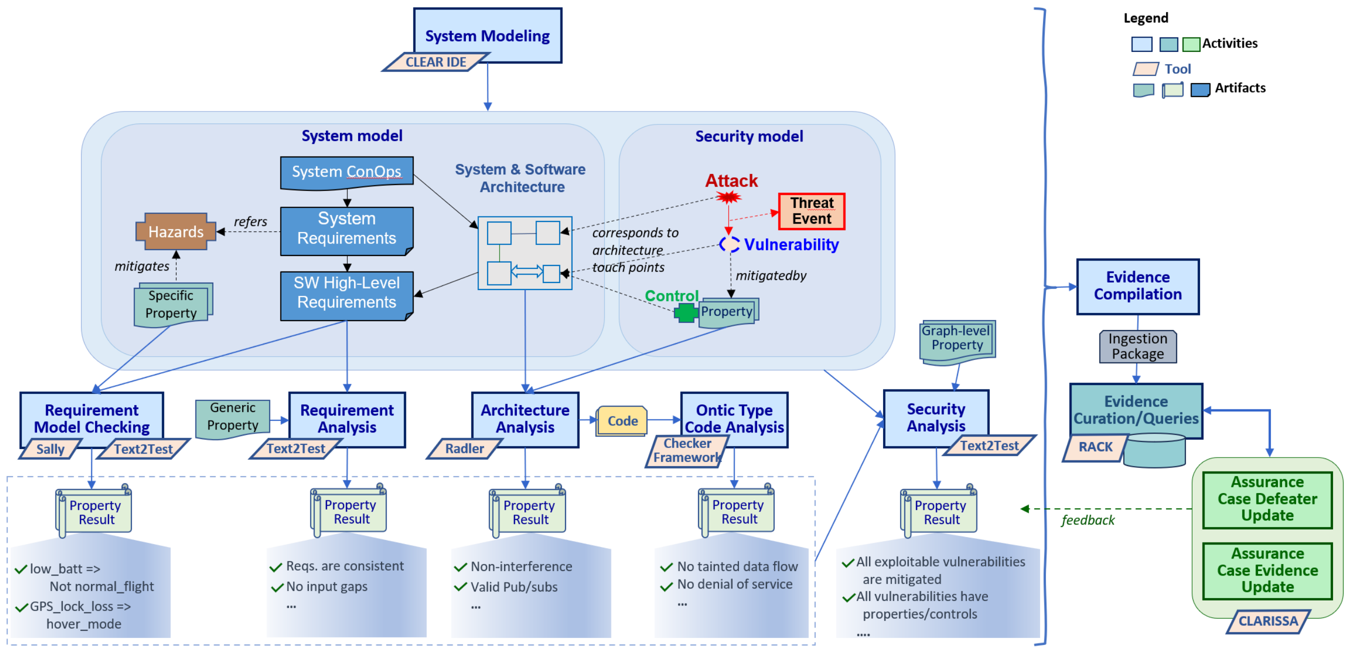


Figure 1: High-level Development, Verification, and Assurance Workflow

Evidence Compilation All evidence artifacts and provenance ontology data from system modeling, analysis and V&V activities is collected and merged in composite set for a complete execution of the activities. The ontology data, properties/results, architecture model, and security model are expressed in CLEAR notation, which enables strong type checking including identification of missing references and mismatches. A mapping is performed from *CLEAR classes* to the *RACK ontology classes* used in further evidence archival and curation. This alignment/mapping of CLEAR with the RACK ontology was essential to be able to ingest the data into the RACK database. Also, such mapping provides flexibility for exporting to other databases as well. An *ingestion package* is created containing all the evidence data in multiple sets of CSV files identified with the RACK ontology classes.

Evidence Curation/Queries We use the RACK [8] semantic triplestore database to curate the evidence produced by the DesCert tools. RACK uses a core ontology that builds upon the W3C standard PROV-S provenance ontology and prescribes specific classes to appropriately identify diverse sets of evidence from different tools and processes into a unified semantic model aimed at Assurance case construction. To support a property-based approach, some of the RACK's core classes were extended as an overlay ontology. A crucial aspect of evidence curation is the ingestion and triplification of the generated evidence into the RACK semantic triplestore. RACK allows the ingestion of evidence into different data graphs, which used as sources of running semantic queries. A salient feature of RACK is the use of nodegroup abstraction [9] that serve as an easy way to construct expressive graph queries over Ontology classes and triples ingested into RACK. Nodegroups make the search for meaningful information in a huge dataset easier to use and analyze.

Assurance Case Construction Once the evidence is generated and ingested into RACK, an assurance case is constructed using this evidence. We use the CAE framework to develop assurance cases advocated by the CLARISSA methodology [10]. The CAE framework proposes the development of an assurance argument by top-level claims supported by sub-claims, side-claims, and evidence. Each of the claims can be challenged be defeaters and the CAE framework allows for the capture of defeaters to claims. To complete the assurance argument, the defeaters must be mitigated using counter-claims. The evidence used in the CAE framework is sourced from the RACK database by using appropriate nodegroups necessary to populate the claims, sub-claims, and defeaters in the assurance case. We used the ASCE tool from Adelard [11] for CAE assurance case construction for the entire ArduCopter AFS safety and security assurance.

ETB2 The workflow in Figure 1 is orchestrated using ETB2, which provides a rigorous framework for the continuous production and maintenance of evidence and assurance arguments. ETB2 allows different tools and services to communicate through a distributed tool bus, using Datalog rules to generate checkable evidence (e.g., files) in support of claims related to updates of evidence. All the artifacts generated by the workflow, along with the provenance ontology data, are exchanged among tools via ETB2.

The ETB2 architecture (Figure 2) is designed for extensibility, assurance, and semantic neutrality. It allows different tools and services to communicate through a distributed tool bus, using Datalog rules to generate checkable evidence (e.g., files) in support of claims. In ETB2, one can add new claims, tools, services, workflows, argument formats, servers,

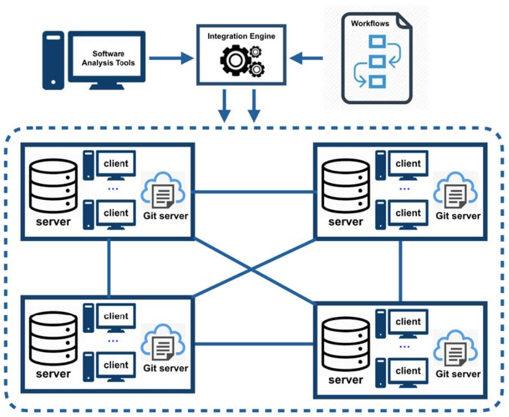


Figure 2: ETB2 Architecture

and clients (extensibility); maintain rigorous, reproducible, and verifiable evidence such that tool integration is not constrained by specific languages, models, or tools.

ETB2 consists of modules that work together to enable an end-to-end, decentralised safety assurance process in which multiple entities are involved to establish safety claims supported by evidence. ETB2 provides a simple abstraction (a workflow) for defining how these entities (also known as services) consume and create evidence, as well as subscribe to other services. Workflows in ETB2 reduce to a subset of Datalog with no negation, distributed top-down left-to-right evaluation, and add uninterpreted and interpreted predicates to the language. Predicates in a workflow capture tool and service invocations or workflow scripts. The data for these predicates include JSON terms as well as files, and tool and session handles. Multiple predicates in a workflow are involved in establishing safety claims supported by evidence. ETB2 nodes run a Datalog engine that can be used to implement workflows integrating different services [12], [13].

3. Arducopter Usecase

The Ardupilot platform [14] is an open and extensively used software suite in industry to control navigation of drones; Arducopter is one of the drone types. In this section, we describe the Arducopter's Advanced Fail Safe (AFS) use case, demonstrating Radler code generation [14], [15]. We consider developing assurance for the Arducopter AFS software runtime monitor that checks the mission geo-fences, GPS/communication losses, and low battery conditions. This open use case has sufficient complexity with comparable functions on an aircraft to demonstrate safety and security assurance workflow in ETB2.

We design the AFS system architecture using Radler, consisting of the logical and physical parts. The logical part (Figure 3) is specified in terms of node and topic similar to ROS. The nodes execute independently and periodically,

1. Nodes, depicted as ellipses, communicate via topics represented as rectangles. The `/mavros` node publishes to `/mavros/battery` topic to be subscribed from the `/gateway` node. Subsequently, the `/gateway` node publishes this information to the `/afs/battery` topic, where it is subscribed by the `/afs_battery` node for using in its step function.

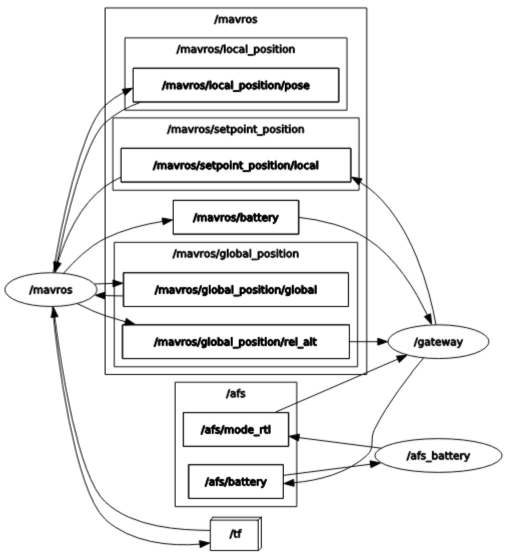

/mavros
/mavros/local_position
/mavros/local_position/pose
/mavros/setpoint_position
/mavros/setpoint_position/local
/mavros/battery
/mavros/global_position
/mavros
/mavros/global_position/global
/mavros/global_position/rel_alt
/gateway
/afs
/afs/mode_rtl
/afs_battery
/afs/battery
/tf

Figure 3: Radler AFS nodes and topics[1]

and subscribe from and publish to topics. Physical parts of AFS map nodes to the processes on a companion computer, which is a separate computer hardware communicating with the Arducopter flight software using MAVLink protocol over an independent channel. The Radler build process takes the architecture definition and individual local functions as inputs and generates executables for the overall system as output. Radler synthesizes glue code for the communication layer based on the logical description and binds it to the source code for node functions to generate a code executable. During the build process, it instruments the system so that platform assumptions such as node periods and channel latencies can be checked at runtime.

In this example, the `/afs_battery` node executes its step function with period of 100 milliseconds. It subscribes from the `/afs/battery` topic to monitor the battery status of the Arducopter and publishes to `/afs/mode_rtl` to initiate a flight mode change. The step function of the `/afs_battery` node is responsible for controlling the flight mode to return to the takeoff location (RTL) when the battery level falls below a certain threshold. The `/gateway` node's step function forwards back-and-forth messages between Radler and ROS worlds. For instance, messages published on the ROS side under the `/mavros/battery` topic are subscribed by the `/gateway` node. Subsequently, the `/gateway` node forwards these messages to the `/afs_battery` node on the Radler side via `/afs/battery`. The `/mavros` node serves as a MAVLink extendable communication node, capable of converting between ROS topics and MAVLink messages. Located on the companion computer, it facilitates communication with flight controllers, ground control stations, and peripherals.

ETB2 Integration We demonstrate how to employ the ETB2 framework for the Radler build of the Arducopter AFS usecase by defining and executing distributed workflows that produce claims supported by evidence. As explained in earlier sections, ETB2 employs Datalog to define workflows

```
radler_build(RADL, WS, PLANTDOT) :- radler_compile(RADL, WS, COMPILE), colcon_build(WS, BUILD),
                                    radler_dump(RADL, WS, DUMP), radler_plantdot(RADL, WS, PLANTDOT).
radler_compile(RADL, WS, RES) :- radler(RADL, WS, 'compile', RES), equals(RES, "compiled").
radler_dump(RADL, WS, RES) :- radler(RADL, WS, 'dump', RES), equals(RES, "jsonized").
radler_plantdot(RADL, WS, RES) :- radler(RADL, WS, 'plantdot', RES), equals(RES, "plantdotted").
colcon_build(WS, RES) :- colcon(WS, RES), equals(RES, "built").
```

Figure 4: ETB2 workflow in Datalog[2]

that utilize services for integrated analysis, verification, and assurance activities. Figure 4 illustrates the Datalog script for our use case. In Figure 4, Radler's core function involves generating codes from the RADL file, i.e., `afs.radl`, transforming them into the ROS2 structure (`radler_compile`). Following this, a standard Colcon build produces the necessary executables (`colcon_build`). After the completion of the Colcon build process, Radler generates a JSON file describing the nodes and topics (`radler_dump`). Additionally, Radler creates a PNG file visualizing the publisher subscriber relationships (`radler_plantdot`).

Once ETB2 is installed, an ETB node can be initialized by running the `etb2 init --path=<nodeSpec>` command, where the `<nodeSpec>` in JSON format contains the necessary information to identify ETB nodes by their IP address and port number. To execute the Datalog description in Figure 4, two services for Radler and Colcon should be added to the ETB2 node with IDs (i.e., radler and colcon) by using the command `etb2 add-service --path=<serviceSpec>`, where the `<serviceSpec>` in JSON format contains the ID, signature, and mode of the service. For example, the Radler service has an ID in the format of `"ID":"radler"`, a signature of `["string","string","string","string"]` and a mode of `["+++-"]`, indicating that the first three arguments (i.e., RADL file name, working directory name, Radler command) are inputs, while the last one is output.

When a service is added, ETB2 auto-generates a JAVA wrapper code for the service. The user is then required to incorporate the intended functionalities into the `run()` method. Once these modifications are made, the wrapper should be recompiled, and ETB2 should be updated with the new version of the service. This update can be done within ETB2 by running the command `etb2 update-service --id=<serviceID>`. Now, a workflow utilizing those services can be integrated into an ETB node using the command `etb2 add-workflow --path=<workflowSpec>`, where the `<workflowSpec>` in JSON format specifies the path to the Datalog script and the possible queries it can answer.

4. Tool-based Evidence Generation

The assurance workflow includes various evidence generation activities for the system model and security model, such as model checking and a set of V&V analysis. These activities are represented in Figure 1 by the rectangular boxes below the models. The input, tool used, and resulting evidence artifacts for each individual evidence generation activity are summarized below:

- *Requirement Model Checking* takes user-specified properties in CLEAR IDE and generating pass/fail/unknown property results and counterexamples, if any, using Text2Test tool [7]. This activity could also be expanded to include model-checking of properties specified and verified using the Sally tool [3], which is suited for extensive and complex temporal system behaviors.
- *Requirement Analysis* via Text2Test verifies a set of general properties over the requirement set itself regarding its consistency, completeness, mode-thrashing, etc. The violation of any general property is considered as a requirement defect, which will be output into the defect report together with the trace to the requirement and counterexamples.
- *Ontic Type Code Analysis* uses a pluggable type checker [16] to check Java code of particular software components for properties such as 'no remote-code execution', 'no null references', 'no tainted data flow'. These properties are used to mitigate specific security vulnerabilities.
- *Graph-level Security Analysis* is the analysis performed in the last step on the entire model – identifying gaps in the reference relationships among ontic objects from the architecture and security models and property result of all the other analysis activities listed above. This analysis proves properties such as 'each vulnerability has an associated property to mitigate it' and 'all exploitable vulnerabilities are mitigated by a property with a Satisfied result'.

5. Evidence Curation Workflow using RACK

ETB2 workflows have been developed to perform RACK Ingestion (see Figure 5) and to query nodegroups to fetch evidence curated in RACK. The RACK Ingestion wrapper consumes a manifest file of an ingestion package, performs ingestion and returns an ETB2 handle, if the ingestion succeeds. The ETB2 handle for RACK ingestion will be subsequently used by an assurance case workflow that pulls evidence from RACK. The RACK query wrapper will use the ETB2 handle from the ingestion phase, the query nodegroup and the datagraph as arguments, and return the results of the query nodegroup in a .csv or text format. The returned csv or text file will be used as a handle for any downstream assurance case construction ETB2 workflows. The basic ingestion and

2. The `radler_build` claim undergoes analysis through user-defined services of Radler and Colcon. The sub-goals are processed from left to right, i.e., `radler_compile`, `colcon_build`, `radler_dump`, and `radler_plantdot`. Upon the satisfaction of `radler_plantdot`, the `radler_build` claim can be added.

3. https://sri-csl.github.io/sally/

querying capabilities for RACK ETB2 workflows given are below. Note that the query workflow can check if an ETB2 claim associated with an ingestion previously succeeded. If no ingestion was performed prior to querying, a new ingestion workflow will be triggered and the nodegroup results will be fetched from RACK. If an ingestion ETB2 claim already exists, then the ETB2 execution will use the ingestion handle, to query RACK. If the ingestion handle did not change from a prior nodegroup query to RACK, the csv or text file handle in the ETB git repo will be returned containing the results of the prior nodegroup execution.

We have also developed ETB2 wrappers that perform consistency checks on the evidence ingested into RACK using RACK ASSIST tools [17]. The RACK ASSIST tools detect trivial errors such as missing description for an ontology class to more severe errors such as an entity's properties violating cardinality constraints specified in the ontology. The RACK ASSIST wrappers will compile the errors in the RACK evidence post-ingestion for an evidence curator to iterate further and fix errors. Consequently, the RACK ASSIST wrappers can provide an option for assurance case tools to only use evidence that have no ontology errors post-ingestion.

```
% Ingestion ETB2 workflow
ingest(Manifest, Status, Errors, Handle) :-
  rack(Manifest, Status, Errors, Handle).
% RACK Query ETB2 workflow
query(Manifest, Nodegroup, Datagraph,
        Status, Errors, Handle) :-
  ingest(Manifest, 'success', IngestHandle),
  queryRack(IngestHandle, Nodegroup,
      Datagraph, Status, Errors Handle).
```

Figure 5: RACK Ingestion Workflow

6. Assurance Case Construction Workflow using CLARISSA methodology and tools

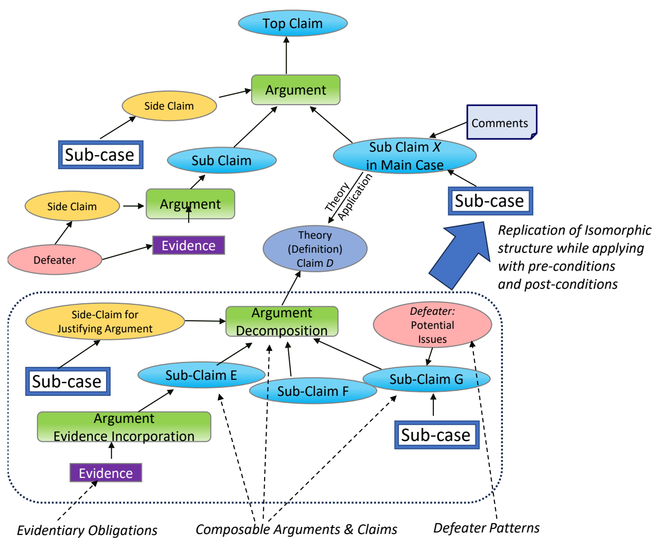


Figure 6: Assurance 2.0 Blocks

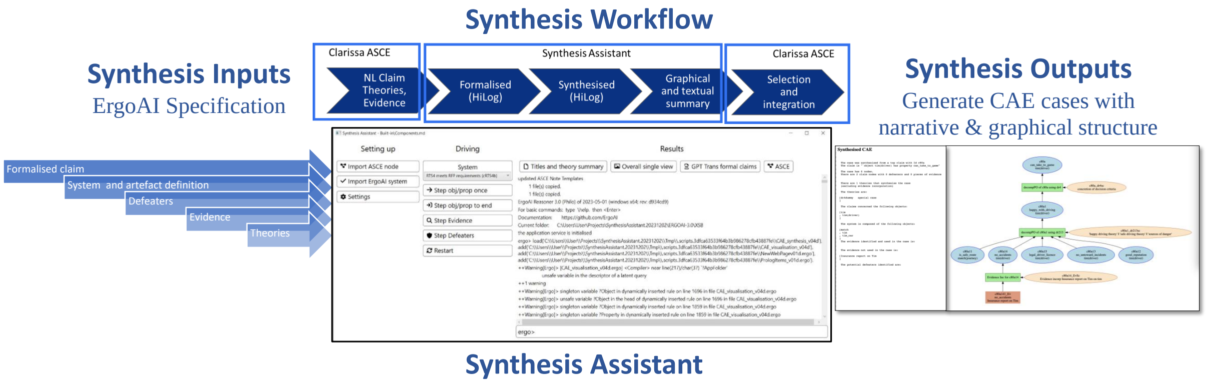
Our assurance case approach is called *Consistent Logical Automated Reasoning for Integrated System Software Assurance (CLARISSA)* – developed for the DARPA Automated Rapid Certification Of Software (ARCOS) program [18]. Assurance cases in CLARISSA follow the approach presented as Assurance 2.0 [19], which builds on the earlier *"Claims, Arguments, Evidence"* or CAE [20] methodology, as shown in the top-left of Figure 6, where the main component of an assurance case is a structured argument represented as a tree of claims linked by argument steps, refines to sub-claims and grounded on evidence. CLARISSA tools are employed to rigorously construct an assurance case utilizing evidence artifacts generated and curated through various tools listed in Sections 4, 5. While detailed discussions of the CLARISSA approach's Assurance 2.0 methodology and the associated tools are provided in [21], [22], this section offers a brief overview of key concepts relevant to a property-driven, theory-based continuous assurance approach.

Property-driven Semantics Analysis with Logic Support: CLARISSA takes a two-step approach, where we first categorically ground the natural language terms used in the descriptions. Assurance cases are automatically transformed into a logical notation, by Assurance and Safety Case Environment ASCE tool [11], that can be subject to various formal analyses at the back end using s(CASP) reasoning engine [23]. The claim/evidence language formalisms, the assurance case transformation to logic programs and subsequent semantic analysis capabilities are detailed in [10], [21].

Fundamentally, assurance cases consist of blocks or "nodes" that describe the properties or relationships applicable to objects in a certain environment. Leveraging this general structure of description, we first categorically specify the terms used in the descriptions – in terms of *objects* of the system, *properties* they possess, and optional *environments* in which the properties of the given objects are valid. e.g., the claim: *"software is correct w.r.t. requirements"* can be expressed in the *object-property-environment* formalism as:

- $object(software)$
- $property(correct\_wrt\_reqs)$
- $environment(env)$
- $claimstmt(software, correct\_wrt\_reqs, env)$

Compositional Assurance with Theories and Defeaters: *Theories* are essentially an *assurance sub-case* [10], [21], defined as a *reusable template* but with *semantics* and *justifications*, that can be applied to various assurance cases. The motivation is that an overall assurance case for a complex system can then mostly be built by instantiating various theories plus integrating arguments. The generalized structure of a theory is shown in Figure 6. The theory *definition* is described under *Claim D* while the theory *application* is instantiated under *Claim X* in the main assurance case in an *isomorphically replicated* fashion. The theory definition can utilize the full breadth of *Claims-Arguments-Evidence (CAE)* blocks available in the CLARISSA ASCE tool [11]. This includes *defeaters* [21] and whose pattern can be introduced at various CAE nodes in the definition of the theory to throw well-known doubts and concerns typically accompanying



```
updated ASCE Note Templates
        1 file(s) copied.
        1 file(s) copied.
ErgoAI Reasoner 3.0 (Philo) of 2023-05-01 (windows x64; rev: d934cd9)
For basic commands: type '\help.' then <Enter>
Documentation:      https://github.com/ErgoAI
Current folder:     C:\Users\User\Projects\SynthesisAssistant.20231202\ERGOAI-3.0\XSB
the application service is initialised
ergo> load('C:\\Users\\User\\Projects\\SynthesisAssistant.20231202\\Tmp\\.scripts.3dfca63533f64b3b986278cfb43887fe\\CAE_synthesis_v04d'),
add('C:\\Users\\User\\Projects\\SynthesisAssistant.20231202\\Tmp\\.scripts.3dfca63533f64b3b986278cfb43887fe\\CAE_visualisation_v04d'),
add('C:\\Users\\User\\Projects\\SynthesisAssistant.20231202\\Tmp\\.scripts.3dfca63533f64b3b986278cfb43887fe\\NewWebPagev01d.ergo'),
add('C:\\Users\\User\\Projects\\SynthesisAssistant.20231202\\Tmp\\.scripts.3dfca63533f64b3b986278cfb43887fe\\PrologItems_v01d.ergo').
++Warning[Ergo]> [CAE_visualisation_v04d.ergo] <Compiler> near line(217)/char(37) `?AppFolder'
            unsafe variable in the descriptor of a latent query
++1 warning
++Warning[Ergo]> singleton variable ?Object in dynamically inserted rule on line 1696 in file CAE_visualisation_v04d.ergo
++Warning[Ergo]> unsafe variable ?Object in the head of dynamically inserted rule on line 1696 in file CAE_visualisation_v04d.ergo
++Warning[Ergo]> singleton variable ?Object in dynamically inserted rule on line 1859 in file CAE_visualisation_v04d.ergo
++Warning[Ergo]> singleton variable ?Property in dynamically inserted rule on line 1859 in file CAE_visualisation_v04d.ergo
ergo>
```

Figure 7: Illustration of Synthesis Assistant

the theory definitions. Thus, any successful application of the defined theories, enforced through the CLARISSA tools, requires satisfaction of various *pre-conditions* and *post-conditions* for each individual theory, which includes *sustainment/refutation* all defeaters, supplying various *evidentiary obligations* and *consistent semantic object-property-environment* specifications [22].

Synthesis Assistant (SA) for Generating Assurance Cases: The synthesis workflow is shown at the top of Figure 7. To generate the CAE structure and assurance case, we need 5 key inputs as shown in the left of Figure 7. First, the basis for the synthesis is a top-level claim in a formalized specification. Second, we need an understanding of what the case is about – i.e, a description of the system and its artifacts. System definitions represent the structure of the system (subsystems, components, hardware, software, etc.) and the associated life-cycle artifacts (code, requirements, tests, etc.) which, might form part of the case. Third, we need the rules for developing the argument steps. In assurance 2.0 these are based on theories, a form of parameterized template that has defined semantics discussed above. In addition, we want to identify possible defeaters to the case and, if there is evidence available, show how it can be incorporated into the case. Finally, with the five synthesis inputs, shown in Figure 7, we can generate cases, providing a graphical CAE and textual summary as synthesis outputs (see [21]).

7. End-to-End integration of DesCert, RACK & CLARISSA into ETB2

In this section, we demonstrate how the ETB2 framework is used to develop and maintain the assurance case for the Arducopter AFS use case, such that incremental changes in artifacts lead to incremental re-evaluation of tool outputs, evidence, and assurance case arguments. Just like how the Radler tool was integrated into ETB2 as a service in Section 3, we also integrate the DesCert tools for evidence generation, RACK tool for evidence curation, and CLARISSA tools for assurance case generation. These services compose to form a workflow that encompasses the activities from requirements and system modeling to assurance case creation.

Figure 1 shows the graphical representation of the workflow for this use case; Figure 8 shows the workflow specification in Datalog. The workflow begins with invoking the `system_modeling` service, which produces various modeling files, requirement files, provenance files, and source code files. These files were manually created by experts, and stored in the version-controlled repository that ETB2 maintains. The `system_modeling` service takes those expert-created files and makes them available as inputs to subsequent tool services. After `system_modeling` executes, the next tool, the `checkerFramework_type_checking` service follows. It takes the properties files, architecture specification files and source code files as input, then performs static analysis on the source code. Likewise, after the `checkerFramework_type_checking` service executes, the following tool runs. The `radler_radl_analysis` service analyzes the architecture files, the `test2Test_requirement_analysis` service analyzes the requirement files, and the `securityAnalysis_and_ingestion_creation` service performs graph analysis for security vulnerabilities. Each of these tools produce property results for the relevant properties defined in the properties files. Finally, in addition to security analysis, the `securityAnalysis_and_ingestion_creation` service also takes the property results files from the previous tools, various modeling and requirement files, and packages them into a RACK ingestion package.

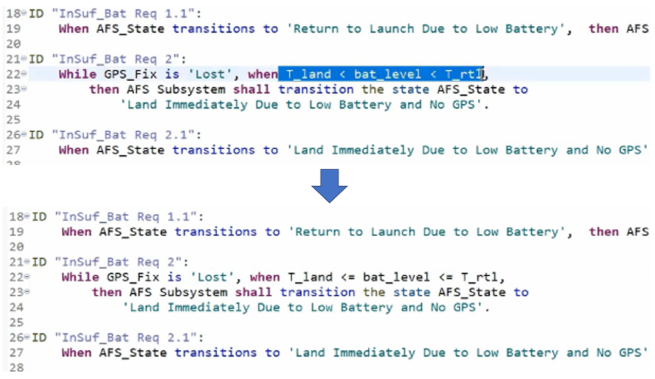
The previous services produce the evidence, and this now needs to be stored and maintained in the RACK. The ingestion package is taken as input to the `rack_ingestion` service, which ingests the data into the RACK and performs a comparison between the existing data in RACK and the new data in the ingestion package. The service returns a string documenting the differences between the old and new data, allowing subsequent services to know what has changed in the data.

4. Workflow describing the integration of DesCert, RACK, and CLARISSA tools to make an end-to-end workflow from design and modeling to assurance case creation.

```
end2end_workflow(Start) :-
 system_modeling(Properties, ArchitectureSpec, SecuritySpec, BaseProcessProvData, RADLProvData, CodeFilesProxy,
      RadlFiles, RequirementProvData, RequirementProperties, RequirementPropertyFiles, RequirementFiles,
      ArchControlPropertyResults),
 checkerFramework_type_checking(Properties, ArchitectureSpec, CodeFilesProxy, CFPropertyResults),
 radler_radl_analysis(Properties, ArchitectureSpec, RadlFiles, RadlerPropertyResults),
 text2Test_requirement_analysis(ArchitectureSpec, BaseProcessProvData, RequirementProperties, RequirementProvData,
      RequirementPropertyFiles, RequirementFiles, ReqAnalysisPropertyResults),
 securityAnalysis_and_ingestion_creation(Properties, ArchitectureSpec, SecuritySpec, BaseProcessProvData, RADLProvData,
       RequirementProvData, RequirementProperties, ArchControlPropertyResults, CFPropertyResults, RadlerPropertyResults,
       ReqAnalysisPropertyResults, IngestionPackageFiles, IngestionPackageManifest),
 rack_ingestion(IngestionPackageFiles, IngestionPackageManifest, SuccessOrError, ErrorMsg, Handle,Diff),
 update_asce_evidence_dnr(Diff, AsceFile),
 detect_property_violation(Diff, PropertyViolated),
 update_asce_with_defeaters(PropertyViolated, AsceFile).
```

Figure 8: ETB2 workflow in Datalog[4]

After evidence has been stored in RACK, the assurance case generation tools can use the output of the `rack_ingestion` service to determine which CLARISSA services should be triggered. The CLARISSA services can then make use of the evidence in the RACK. ASCE is a graphical tool that assists the expert-guided creation of an assurance case that is made up of claim, argument, and evidence nodes. There are many ways to compose these building blocks to create an assurance case, but in the example, the top-most claim is decomposed via a decomposition argument into subclaims, and each subclaim is supported by an evidence node via an integration argument. These evidence nodes can contain cached information queried from RACK. Just as the various modeling, requirement, and code files were made available by the `system_modeling` service, the ASCE assurance case was also created by experts and made available by the `update_asce_evidence_dnr` service for subsequent services to use. However, before the service is finished executing and produces the ASCE file, it checks the output of the `rack_ingestion` service. If there have been changes, then the service queries RACK and updates the RACK evidence cached in the ASCE file. This ensures that the evidence in the assurance case is up to date. The `detect_property_violation` service executes next, and determines from the output of the `rack_ingestion` service whether any of the properties have been violated. It generates an output string that tells the following service whether to execute. If there are no violated properties, then the `update_asce_with_defeaters` service remains dormant and this concludes the execution of the workflow. However, if there are violated properties, then the `update_asce_with_defeaters` service executes. In ASCE, an assurance case contains many claim nodes. Any of these claim nodes can be called into question by a defeater node, which invalidates the claim. For a case to be considered as valid, all of the defeater nodes need to be evaluated and addressed. If a claim asserts a property but this property is later found by evidence to be violated, then a defeater node can be attached to the claim to invalidate it. The `update_asce_with_defeaters` service uses the Synthesis Assistant (SA) tool, which generates an assurance case fragment based on certain rules about theories, claims, and evidence, to generate a defeater subcase. The subcase is then merged with the main ASCE case, and attached to the claim whose property is violated. Using this workflow, we developed two scenarios to demonstrate how changes in artifacts from early in the workflow can propagate to change the artifacts at the end of the workflow.



```
ID "InSuf_Bat Req 1.1":
    When AFS_State transitions to 'Return to Launch Due to Low Battery',  then AFS

ID "InSuf_Bat Req 2":
    While GPS_Fix is 'Lost', when T_land < bat_level < T_rtl,
        then AFS Subsystem shall transition the state AFS_State to
            'Land Immediately Due to Low Battery and No GPS'.

ID "InSuf_Bat Req 2.1":
    When AFS_State transitions to 'Land Immediately Due to Low Battery and No GPS'
```

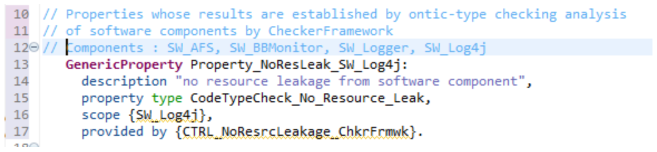
```
ID "InSuf_Bat Req 1.1":
    When AFS_State transitions to 'Return to Launch Due to Low Battery',  then AFS

ID "InSuf_Bat Req 2":
    While GPS_Fix is 'Lost', when T_land <= bat_level <= T_rtl,
        then AFS Subsystem shall transition the state AFS_State to
            'Land Immediately Due to Low Battery and No GPS'.

ID "InSuf_Bat Req 2.1":
    When AFS_State transitions to 'Land Immediately Due to Low Battery and No GPS'
```

Figure 9: Scenario 1: Requirement modifications[5]

Scenario 1: Changes in Requirements - We start by assuming that we have already created an initial set of requirements and system models, performed analysis on the files, ingested the artifacts from the analyses into the RACK, and developed an initially valid assurance case that references the artifacts. Now we make a trivial change in the requirements as shown in Figure 9. The requirement statement for what happens when GPS is lost and the battery level is between 2 thresholds is changed so that there is no impact on the resulting properties. However, since there is a change in requirements, requirement analysis needs to be rerun and the result artifacts are updated.

Once this change is committed, ETB2 will detect a change in the contents of the file. It identifies the services that are affected by this change and need to be updated so that the overall workflow is up to date. First, the `test2Test_requirement_analysis` service that takes the requirement files as input is run on the changed requirement files. The `securityAnalysis_and_ingestion_creation` service takes the updated analysis artifacts and packages them together with the

5. Changes the condition for what happens when GPS is lost and the battery level is between 2 thresholds.



```
// Properties whose results are established by ontic-type checking analysis
// of software components by CheckerFramework
// Components : SW_AFS, SW_BBMonitor, SW_Logger, SW_Log4j
   GenericProperty Property_NoResLeak_SW_Log4j:
     description "no resource leakage from software component",
     property type CodeTypeCheck_No_Resource_Leak,
     scope {SW_Log4j},
     provided by {CTRL_NoResrcLeakage_ChkrFrmwk}.
```

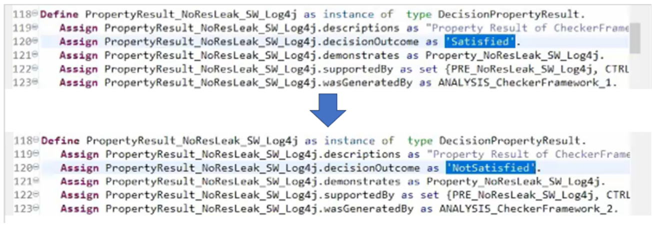
Figure 10: “No Resource Leak” property definition[6]

unchanged artifacts from other services. The `rack_ingestion` service then ingests the new package into RACK. Since the artifacts have changed, the data stored in RACK is changed as well, and the RACK service outputs a log of the changes. These changes are parsed by the `update_asce_evidence_dnr` service, which determines that the DNRs in the assurance case need to be updated. It calls the command line interface to ASCE to update the case. At this point, the case is updated with the latest requirements update. No properties were determined to be invalidated by the upstream analysis services, and so the pre-update assurance case and associated claims are still valid. The assurance case is simply updated with the latest evidence based on associated analysis run by ETB2.

Scenario 2: Changes in Software Code - We begin with the same initial assumption as scenario 1. Instead of changing the requirements, we now make an update to the source code that introduces a dependency on a log4j library that is known to have a security vulnerability. Just like in scenario 1, after the source code is committed, ETB2 will detect a change in the files and identify the services that depend on the files. In this case, they are the `checkerFramework_type_checking` service followed by the `securityAnalysis_and_ingestion_creation` service, and so they will both rerun to update their output. Because the source code now introduces a known vulnerability, the result of the analysis will conclude that one of the security properties is violated. This property is specified in CLEAR notation and provided to the `checkerFramework_type_checking` service in the ”Properties” argument. Figure 10 shows how the property is defined - it includes the type of property it is, what scope of the code the property covers, and other information. A property result is associated with a property, and can have the decision outcome as ”Satisfied” or ”NotSatisfied”. If the decision outcome is ”NotSatisfied”, then the property that the result is associated with is considered to be invalidated. Figure 11 shows the change in Checker Framework analysis results due to the updated source code, which invalidates the log4j *“no resource leak”* property.

Like scenario 1, the updated analysis artifacts are packaged and then ingested into RACK. The `rack_ingestion` service will output a log of the changes, including the invalidation of the no resource leak property. This log is parsed by the `detect_property_violation` service - since there is a change in property result, the `update_asce_with_defeaters` service runs and generates an assurance case fragment from the claim node invalidated

6. The definition of the no resource leak property associated with the log4j library, used by the Checker Framework tool to produce property results the analysis.



```
Define PropertyResult_NoResLeak_SW_Log4j as instance of  type DecisionPropertyResult.
   Assign PropertyResult_NoResLeak_SW_Log4j.descriptions as "Property Result of CheckerFrame
   Assign PropertyResult_NoResLeak_SW_Log4j.decisionOutcome as 'Satisfied'.
   Assign PropertyResult_NoResLeak_SW_Log4j.demonstrates as Property_NoResLeak_SW_Log4j.
   Assign PropertyResult_NoResLeak_SW_Log4j.supportedBy as set {PRE_NoResLeak_SW_Log4j, CTRL
   Assign PropertyResult_NoResLeak_SW_Log4j.wasGeneratedBy as ANALYSIS_CheckerFramework_1.
```

```
Define PropertyResult_NoResLeak_SW_Log4j as instance of  type DecisionPropertyResult.
   Assign PropertyResult_NoResLeak_SW_Log4j.descriptions as "Property Result of CheckerFrame
   Assign PropertyResult_NoResLeak_SW_Log4j.decisionOutcome as 'NotSatisfied'.
   Assign PropertyResult_NoResLeak_SW_Log4j.demonstrates as Property_NoResLeak_SW_Log4j.
   Assign PropertyResult_NoResLeak_SW_Log4j.supportedBy as set {PRE_NoResLeak_SW_Log4j, CTRL
   Assign PropertyResult_NoResLeak_SW_Log4j.wasGeneratedBy as ANALYSIS_CheckerFramework_2.
```

Figure 11: Scenario 2 changing property results[7]

by the changed property. This assurance case fragment is a defeater case that is merged with the main assurance case by attaching to that claim. Figure 12 shows a zoomed-in portion of the original main assurance case on the left, and then on the right, the same portion after the defeater argument fragment has been merged. After this point, the workflow is concluded, and s(CASP) can be used to show that this updated assurance case is not valid.

In both scenarios, we made a change to upstream artifacts – the requirements documents and the source code. In the development process, both requirements and source code are expected to change frequently. The CI/CD/CA process through ETB2 uses automation to integrate incremental changes to these files with the final assurance case to make sure that artifacts throughout the workflow are kept up to date with the changes. In scenario 1, the changes in requirements did not affect the validity of the assurance case, but it changed the contents of the evidence. As a result of the change, a change in assurance case was made to update the DNRs with the changed evidence. In scenario 2, the changes in source code led to the introduction of a known security vulnerability, and a change in analysis result that in-validated a security property. As a result, a defeater was introduced to the assurance case that made the case invalid.

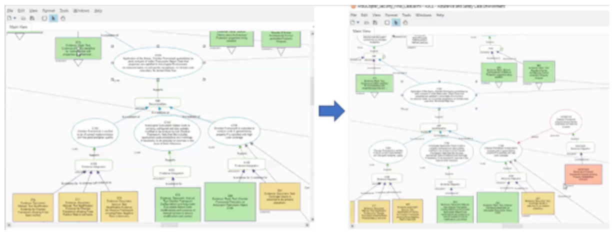


Figure 12: Arducopter AFS assurance case[8]

8. Conclusion and Future work

We have demonstrated the methodology of how assurance cases can be constructed and maintained continuously using the Evidential Tool Bus. We have integrated diverse methodologies for evidence generation, curation and assurance case

7. Change in Checker Framework property results due to an update of the source code that introduced a security vulnerability. Checker Framework analyzes the source code and invalidates the no resource leak property.

8. A zoomed-in section of the Arducopter AFS assurance case in ASCE, showing the update to the assurance case. On the left, before the source code change, the subclaim and argument tree is free from defeaters. On the right, after the source code change, the subclaim and argument tree has an attached defeater introduced by the invalidation of the security property and the execution of the tool services.

construction into our ETB workflows. Such an integrated approach will provide all evidence needed typically for a *Designated Engineering Representative* (DER) about how requirements were specified, how the system and software was designed and developed and so forth into unified sets of evidential claims. All of the V&V activities can be tracked over the life of a project's execution including their evolution. More often, a realistic project involves tool usage under several configurations and ETB workflows explicitly capture configuration management with attestations so that changes to configurations can be understood and their impact on assurance can be maintained and tracked. We demonstrated that ETB can orchestrate the flow of requirements, ontology, and model information along with safety and security analysis results from several formal analyses tools under the DesCert methodology for an industrial Arducopter use case. We also demonstrated continuous assurance using the integrated workflow use case scenario of the safety and security case impacted upon a change to upstream software code.

As part of future work, ETB can capture more complete workflows aimed at certification. Further, if assurance case construction misses some critical evidence, the missing evidence workflows can be integrated into ETB itself as opposed to a restart of the pipeline from upstream tools. The RACK database can be used to bi-directionally communicate semantic information about the status of the evidence used and evidence changes between evidence producers and consumers through ETB. ETB workflows can also generate continuous compliance reports against standards such as DO-178C as part of assurance case evidence. Certification use cases and workflows specific to aerospace-specific *development assurance levels* (DAL) need to be identified and captured in ETB. We leave these explorations for future work.

Acknowledgements & Disclaimer Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This research was developed with funding from the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-20-C-0226. The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] N. Shankar, D. Bhatt, M. Ernst, M. Kim, S. Varadarajan, S. Millstein, J. Navas, J. Biatek, H. Sanchez, A. Murugesan *et al.*, "Descert: Design for certification," *arXiv preprint arXiv:2203.15178*, 2022.

[2] C. Hubbs and J. Myren, "Automating airborne software certification compliance using cert devops," in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*. IEEE, 2023, pp. 1–6.

[3] J. L. de La Vara, M. Borg, K. Wnuk, and L. Moonen, "An industrial survey of safety evidence change impact analysis practice," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, 2016.

[4] M. Gleirscher, S. Foster, and Y. Nemouchi, "Evolution of formal model-based assurance cases for autonomous robots," in *17th Int'l Conf. on Software Engineering and Formal Methods (SEFM), Proceedings 17*. Springer, 2019, pp. 87–104.

[5] S. Bensalem, P. Katsaros, D. Ničković, B. H.-C. Liao, R. R. Nolasco, M. A. E. S. Ahmed, T. A. Beyene, F. Cano, A. Delacourt, H. Esen *et al.*, "Continuous engineering for trustworthy learning-enabled autonomous systems," in *International Conference on Bridging the Gap between AI and Reality*. Springer, 2023, pp. 256–278.

[6] L. Sorokin, R. Bouchekir, T. A. Beyene, B. H.-C. Liao, and A. Molin, "Towards continuous assurance case creation for ads with the evidential tool bus," in *European Dependable Computing Conference*. Springer, 2024, pp. 49–61.

[7] D. Bhatt, H. Ren, A. Murugesan, J. Biatek, S. Varadarajan, and N. Shankar, "Requirements-driven model checking and test generation for comprehensive verification," in *NASA Formal Methods Symposium*. Springer, 2022, pp. 576–596.

[8] P. Cuddihy, D. Russell *et al.*, "Aviation certification powered by the semantic web stack," in *The Semantic Web – ISWC 2023*. Cham: Springer Nature Switzerland, 2023, pp. 345–361.

[9] V. S. Kumar, P. Cuddihy, and K. S. Aggour, "Nodegroup: a knowledge-driven data management abstraction for industrial machine learning," in *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning*, 2019, pp. 1–4.

[10] S. Varadarajan, R. Bloomfield, J. Rushby, G. Gupta, A. Murugesan, R. Stroud, K. Netkachova, and I. Wong, "Clarissa: Foundations, tools & automation for assurance cases," in *42nd AIAA/IEEE Digital Avionics Systems Conference (DASC)*, October 2023.

[11] Adelard LLP, "Assurance and safety case environment (asce), http://www.adelard.com/asce," 2024, accessed on June 20, 2023. [Online]. Available: http://www.adelard.com/asce

[12] S. Cruanes, G. Hamon, S. Owre, and N. Shankar, "Tool integration with the evidential tool bus." in *VMCAI*, vol. 7737. Springer, 2013.

[13] S. Cruanes, S. Heymans, I. Mason, S. Owre, and N. Shankar, "The semantics of datalog for the evidential tool bus," *Specification, Algebra, and Software: Essays Dedicated to Kokichi Futatsugi*, 2014.

[14] ArduPilot Site. [Online]. Available: https://ardupilot.org/dev

[15] Radler AFS Use case. [Online]. Available: https://github.com/SRI-CSL/radler/tree/ros2/examples/ardupilot

[16] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller, "Building and using pluggable type-checkers," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 681–690.

[17] GE High Assurance Software. RACK ASSIST. [Online]. Available: https://github.com/ge-high-assurance/RACK/wiki/Data-Verification

[18] Defense Advanced Research Projects Agency (DARPA). Automated Rapid Certification Of Software (ARCOS). [Online]. Available: https://www.darpa.mil/program/automated-rapid-certification-of-software

[19] R. Bloomfield and J. Rushby, "Assessing confidence with assurance 2.0," *arXiv preprint arXiv:2205.04522*, 2024. [Online]. Available: https://arxiv.org/abs/2205.04522v4

[20] Adelard. Claims Arguments Evidence (CAE). [Online]. Available: https://claimsargumentsevidence.org/

[21] S. Varadarajan, R. Bloomfield, J. Rushby, G. Gupta, A. Murugesan, R. Stroud, K. Netkachova, and I. Wong, "Consistent Logical Automated Reasoning for Integrated System Software Assurance (CLARISSA), DARPA ARCOS Final Report. To appear shortly," Tech. Rep., June 2024.

[22] S. Varadarajan, R. Bloomfield, J. Rushby, G. Gupta, A. Murugesan, R. Stroud, K. Netkachova, I. Wong, and J. JArias, "Enabling theory-based continuous assurance: A coherent approach with semantics and automated synthesis," in *11th Int'l Workshop on Next Generation of System Assurance Approaches for Critical Systems (SASSUR), SAFECOMP - 43rd Int'l Conf. on Computer Safety, Reliability and Security*, September 2024.

[23] A. Murugesan, I. H. Wong, S. Varadarajan, J. Arias, E. Salazar, G. Gupta, R. Stroud, R. Bloomfield, and J. Rushby, "Automating Semantic Analysis of System Assurance Cases using Goal-directed ASP," *Submitted to Int'l Conf. on Logic Programming (ICLP)*, 2024.