Aviation Certification Powered by the Semantic Web Stack

Paul Cuddihy[1], Daniel Russell[2], Eric Mertens[3],
Kit Siu[1], Dave Archer[3], Jenny Williams[1]

[1] GE Aerospace Research, Niskayuna, NY 12309, USA
[2] GE Aerospace, Grand Rapids, MI 49512, USA
{cuddihy, daniel.russell, siu, weisenje}@ge.com
[3] Galois, Portland, OR, 97204 USA
{dwa, emertens}@galois.com

Abstract. Every deployed DoD system undergoes certification (or qualification, for military) to assess the software system's fitness for use. Certification requires that human subject matter expert look over evidence and evaluate its conformance to standards such as DO-178C or the Risk Management Framework (RMF). Current practices are not keeping pace with the ever-increasing size of software systems and the amount of evidence required for their certification. This problem is further exasperated when platforms are comprised of systems of systems developed by multiple suppliers, each providing data generated by different tools, in different formats, and captured with different granularity. We demonstrate the application of W3C Semantic web technologies to perform efficient evidence curation under a military research program. This tech stack offers the right solutions for integrating data from heterogeneous sources and for performing graph-traversal queries across data that changes at a regular frequency.

Keywords: Recursive graph traversal, certification, curation.

1 Introduction

Every deployed aviation system undergoes certification (or qualification, for military) to assess the software system's fitness for use; whether it's following DO-178C guidance, a Certification and Accreditation (C&A) process, Risk Management Framework (RMF), or Assessment and Authorization (A&A). Certification requires that a human subject matter expert look over evidence and evaluate its conformance to standards. With evidence consisting of ever-increasing numbers of requirements, architecture components, test cases, and additional data, the current practices are not able to scale. This problem is further exasperated when platforms are made up of systems of systems developed by multiple suppliers, each providing data generated by different tools, in different formats, and captured with different levels of granularity.

One way to solve this problem is through efficient evidence curation. Curation normalizes the data by defining a schema and organizing the artifacts against it. It must

then provide easy access to the data so that it can be used to develop a certification compliance report or to assemble an assurance case with a structured argument showing how the system meets the goals of safe and secure operation. These reports and assurance cases may require updating at a regular frequency to reflect software updates and bug fixes.

The W3C Semantic Web stack offers the right solution for integrating data from heterogeneous sources. An ontology provides the means to reify a schema – it provides the rigidity needed to define an ontology with common subjects and predicates universally applicable to certification evidence. Inference and ingestion tools provide the means to check the dataset's compliance to the ontology either during or post ingestion. SPARQL is an excellent match for the inherently recursive graph structure of assurance case evidence and can provide results for a certification report.

In this paper, we describe how we curate certification evidence into our open source Rapid Assurance Curation Kit (RACK), which consists of a semantic triplestore backed by an ontology. This curation platform is developed under DARPA Automated Rapid Certification of Software (ARCOS) [1]. There are three performer teams on this research program that generate evidence (known as Technical Area 1—TA1) and three other performer teams that use the evidence to assemble assurance cases (known as TA3). The development of RACK and the curation effort is known as TA2; the authors of this paper are part of this sole performer team on the program. RACK is used by all performer teams on ARCOS.

2 Background

2.1 Definition of Data Curation

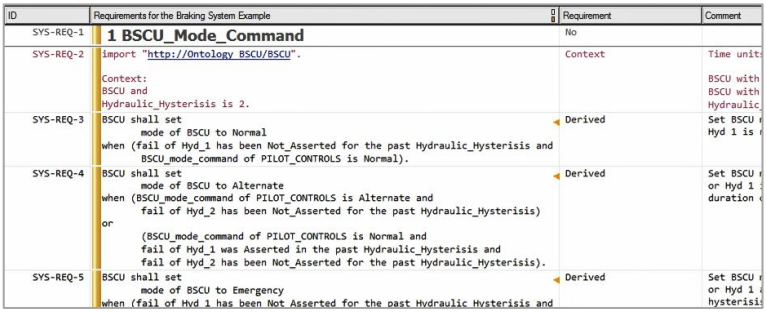
Curation means organizing the data, assessing its quality, and providing easy access to it. A system development process produces artifacts in a multitude of formats, some of which we will illustrate in the next subsection. To make use of these artifacts, the data within must be identified, extracted, and organized. To ensure that the data is curated correctly, data verification is performed at various stages. Verification is done at ingestion time to make sure that all incoming data matches the types specified by the ontology and that all properties and classes are correct. The next level of verification is performed after ingestion of the complete dataset to ensure that it adheres to any specified ontology constraints such as cardinality. The final verification step is domain specific. In airborne system certification, for example, we perform a query that counts how many tests do not trace to requirements.

2.2 Diversity of Tools and Artifacts

The system development process is typically defined in a series of planning documents that describe how the development team will conform to the certification guidelines. Examples are PSAC (Plan for Software Aspects of Certification), SDP (Software Development Plan), SVP (Software Verification Plan), SCMP (Software Configuration

Management Plan), and SQAP (Software Quality Assurance Plan). These plans are freeform text documents; an example template can be found here [2]. Planning documents describe how the development team will generate artifacts through a series of development activities and how those artifacts will be used as evidence that the team has followed the plan. A system development process often uses multiple tools that produce a diversity of artifacts in various formats. RACK would curate the elements of the plans and the evidence produced by development activities, and show how evidence is related, revealing where the development process may have deficiencies or errors.

IBM Rational® DOORS® is a requirements management tool [3] where members of an organization can access, author, and modify requirements. The tool provides links to other DOORS® objects, but the plans dictate traceability to design items such as source code or testing. In this work, RACK would curate from DOORS® the requirement identifier, the requirement text, and any traceability information (**Fig. 1**).



ID	Requirements for the Braking System Example	Requirement	Comment
SYS-REQ-1	**1 BSCU_Mode_Command**	No	
SYS-REQ-2	import "http://Ontology_BSCU/BSCU". Context: BSCU and Hydraulic_Hysterisis is 2.	Context	Time unit: BSCU with BSCU with Hydraulic
SYS-REQ-3	BSCU shall set mode of BSCU to Normal when (fail of Hyd_1 has been Not_Asserted for the past Hydraulic_Hysterisis and BSCU_mode_command of PILOT_CONTROLS is Normal).	Derived	Set BSCU Hyd 1 is
SYS-REQ-4	BSCU shall set mode of BSCU to Alternate when (BSCU_mode_command of PILOT_CONTROLS is Alternate and fail of Hyd_2 has been Not_Asserted for the past Hydraulic_Hysterisis) or (BSCU_mode_command of PILOT_CONTROLS is Normal and fail of Hyd_1 was Asserted in the past Hydraulic_Hysterisis and fail of Hyd_2 has been Not_Asserted for the past Hydraulic_Hysterisis).	Derived	Set BSCU or Hyd 1 duration
SYS-REQ-5	BSCU shall set mode of BSCU to Emergency when (fail of Hyd_1 has been Not_Asserted for the past Hydraulic_Hysterisis and	Derived	Set BSCU or Hyd 1 hysterisi

Fig. 1. Illustration of requirements captured in DOORS®[1]

Developing requirements-based tests is an important verification step, often captured in XML. **Fig. 2** illustrates a test procedure file that would be created for each requirement. In this work, RACK would curate evidence information such as test identifier and the requirement that this test verifies. After tests are run, results could be saved in a myriad of possible formats depending on the platform. Since it's commonplace for test procedures to define expected outputs, RACK would curate the pass/fail result from each test output file.

```
<ctpm:TestStep>
    <ctpm:UniqueId>SYS-REQ-6_Context_1_TS026</ctpm:UniqueId>
    <ctpm:RequirementId>SYS-REQ-6</ctpm:RequirementId>
    <ctpm:ContextId>Context_1</ctpm:ContextId>
    <ctpm:TestCaseId>SYS-REQ-6-TC014_1</ctpm:TestCaseId>
    <ctpm:TestStepId>TS026</ctpm:TestStepId>
    <ctpm:InputData>
        <ctpm:SpecificationName>BSCU__pressure__BSCU__Hyd_1__Properties__hy
        <ctpm:TestData>0</ctpm:TestData>
    </ctpm:InputData>
    <ctpm:InputData>
        <ctpm:SpecificationName>BSCU__threshold__BSCU__Hyd_1__Properties__h
        <ctpm:TestData>2000</ctpm:TestData>
    </ctpm:InputData>
```

Fig. 2. Illustration of a test procedure in XML format

[1] Requirements are textual in nature. This illustrative example is from our 2019 SAE paper [4] where we introduced a tool to capture requirements in a more formal and structural way.

Many other tools exist that aid in developing artifacts within the development process. RACK is a common repository for evidence from all tools. By combining evidence produced from the multitude of tools, a knowledge graph can be assembled that allows for the analysis of the entire development process from one place.

3 Application of the Semantic Web Stack

There are two features of the Semantic Web stack that influenced us to select it for this certification challenge. First, the graph view and recursive graph query operators in SPARQL are an intuitive fit to evidence curation. Second, the ability to write an ontology with natural support for subclasses, sub-properties, and cardinality constraints, and to write inferences to ensure model compliance were a more natural fit than a relational schema.

3.1 Graph Databases and SPARQL

Common to all certification guidance is the need to define requirements with success and failure criteria. A graph display of a subset of curated evidence is shown in **Fig. 3**. In this example, a SYSTEM component is governed by a PIDS_Req (Prime Item Development requirement) which is satisfied by additional requirements such as SRS_Req (Software Requirements Specification). Some requirements are verified by tests, which may also verify other types of requirements. Some tests are confirmed by test results. Both the subclassing of REQUIREMENT and the visual graph structure are very intuitive to the certification community. Importantly, this type of graph display is very close to the natural state of the data in a triplestore, indicating a strong fit.

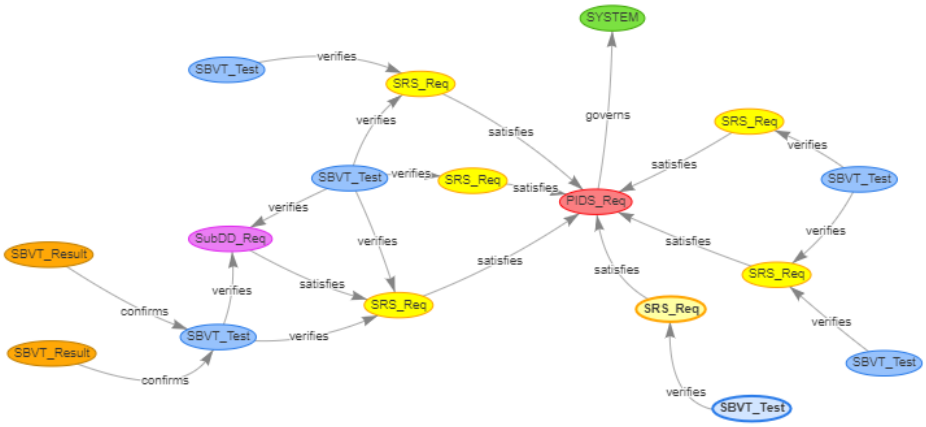


Fig. 3. Graph view of curated evidence

In most examples, system components are arranged in a tree-like network of *partOf* relationships and requirements can *satisfy* layers of additional requirements, creating a need for straight-forward ways to write recursive graph queries. SPARQL's graph

structure and recursive operators such as * and + are a natural fit. Take for example this simple query:

```
prefix TESTING:<http://arcos.rack/TESTING#>
prefix SYSTEM:<http://arcos.rack/SYSTEM#>
prefix REQUIREMENTS: <http://arcos.rack/REQUIREMENTS#>

select distinct ? testStatus where {{
        ? testStatus a TESTING:TEST_STATUS .
        <SOME_URI> (^(SYSTEM:partOf) |
                    ^(REQUIREMENTS:governs) |
                    ^(REQUIREMENTS:satisfies) |
                    ^(TESTING:verifies) |
                    ^(TESTING:confirms) |
                      TESTING:result    )+ ?testStatus .
        }}
```

The URI of any instance of any subclass* of SYSTEM, REQUIREMENT, TEST, or TEST_RESULT may be substituted in for <*SOME_URI*> and the query will return a list of test statuses (*Passed, Failed, Indeterminate)* that roll up the piece of evidence. This SPARQL query will find the test status through layers of subsystems, layers of requirements, and all associated tests. The ability of such a relatively simple SPARQL query to deliver powerful results for a variety of types of evidence is a clear indication that the W3C Semantic Web stack is a great fit for our domain.

3.2 Authoring the Ontology

The ontology is written using the Semantic Application Design Language (SADL) [5] [6]. SADL is an open-source, controlled-English language that is automatically converted to OWL [7]. Besides being an ontology language, SADL is an Eclipse-integrated development environment (IDE) with Xtext [8]. This environment provides semantic coloring of different types of concepts in models, hyperlinking of concepts to their definitions and usage, graphical visualization of models, type checking, content assist, and other functionality useful for authoring models. Below is a snippet of the ontology written in SADL with the OWL translation below. Notice how SADL is very natural for humans to read and write as compared to OWL, even with its standard serializations. This allows non-semantic experts to read, write, and contribute to the data model, which is important for maintainability, as we will describe in the next section on the participative process of updating and maintaining this model throughout the ARCOS program. Our ontology consists of approximately 7,800 lines of SADL, which compiles into 7,400 triples. The semantic model in its entirety can be found on GitHub[2]. A more in-depth discussion of the model can also be found in [9]. The snippet we included below does not demonstrate cardinality constraints but note that this can be done inline in the SADL model.

[2] https://github.com/ge-high-assurance/RACK/tree/master/RACK-Ontology/ontology

```
REQUIREMENT
  (note "Captures (both high- and low-level) properties of a process or arti-
  fact that are to be assessed")
  is a type of ENTITY.
  governs (note "ENTITY(s) that are the subject of the requirement")
  describes REQUIREMENT with values of type ENTITY.
  governs is a type of wasImpactedBy.
  satisfies (note "Parent ENTITY(s) (e.g. REQUIREMENT) that this REQUIREMENT
  is derived from")
  describes REQUIREMENT with values of type ENTITY.
  satisfies is a type of wasImpactedBy.
  Rq:mitigates (note "ENTITY(s) (e.g. HAZARD) that is being mitigated by this
  REQUIREMENT")
  describes REQUIREMENT with values of type ENTITY.
  Rq:mitigates is a type of wasImpactedBy.
  wasGeneratedBy of REQUIREMENT only has values of type
  REQUIREMENT_DEVELOPMENT.
REQUIREMENT_DEVELOPMENT
   (note "ACTIVITY that produces REQUIREMENTs") is a type of ACTIVITY.
```

```
<owl:Class rdf:about="http://arcos.rack/REQUIREMENTS#REQUIREMENT">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:allValuesFrom>
    <owl:Class rdf:about="http://ar-
      cos.rack/REQUIREMENTS#REQUIREMENT_DEVELOPMENT"/>
   </owl:allValuesFrom>
   <owl:onProperty rdf:resource="http://arcos.rack/PROV-S#wasGeneratedBy"/>
  </owl:Restriction>
 </rdfs:subClassOf>
 <rdfs:comment xml:lang="en">Captures (both high- and low-level) properties of
  a process or artifact that are to be assessed</rdfs:comment>
 <rdfs:subClassOf rdf:resource="http://arcos.rack/PROV-S#ENTITY"/>
</owl:Class>
<owl:Class rdf:about="http://arcos.rack/REQUIREMENTS#REQUIREMENT_DEVELOPMENT">
 <rdfs:comment xml:lang="en">ACTIVITY that produces REQUIREMENTs</rdfs:com-
  ment>
 <rdfs:subClassOf rdf:resource="http://arcos.rack/PROV-S#ACTIVITY"/>
</owl:Class>
<owl:ObjectProperty rdf:about="http://arcos.rack/REQUIREMENTS#governs">
 <rdfs:subPropertyOf rdf:resource="http://arcos.rack/PROV-S#wasImpactedBy"/>
 <rdfs:comment xml:lang="en">ENTITY(s) that are the subject of the require-
  ment</rdfs:comment>
   <rdfs:range rdf:resource="http://arcos.rack/PROV-S#ENTITY"/>
   <rdfs:domain rdf:resource="http://arcos.rack/REQUIREMENTS#REQUIREMENT"/>
</owl:ObjectProperty>
...
```

3.3 Maintaining the Ontology

To sustain the ontology, we formed a participative process where all ARCOS performers are stakeholders in evolving the model. The Data Model Decision Team (DMDT) is made up of one representative from each performer team. The group meets regularly (bi-weekly) and any member of the DMDT can propose a change. We follow a RAPID process as shown in **Fig. 4**. The TA2 team administers the meetings, implements agreed

changes, and maintains the ontology. SADL's role as the key enabler to bringing the Semantic Web stack to this DARPA community cannot be overemphasized. We could not have had as deep discussions, or achieved as good of a consensus, if we were looking at relational schemas, or OWL for that matter.

Fig. 4. We use a participative RAPID approach to maintain the ontology.

4 Evidence Ingestion Pipeline

Ingesting thousands of pieces of various types of evidence from multiple providers across large teams requires a rich set of tools layered over the triplestore and organized into an ingestion pipeline. **Fig. 5** provides a simplified view of this pipeline.

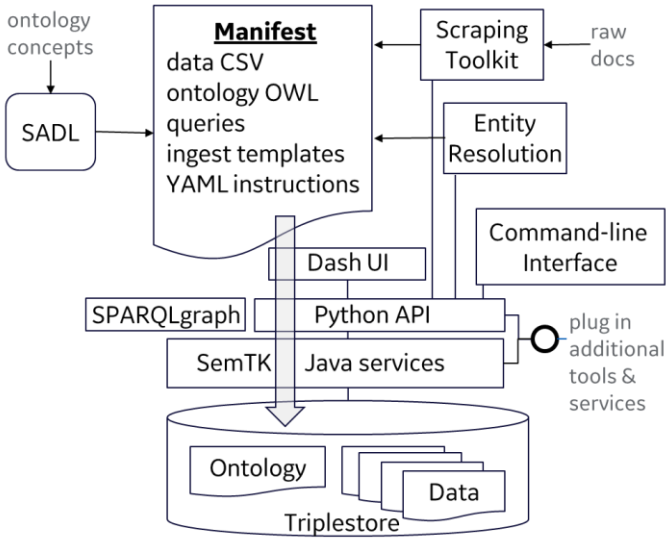


Fig. 5. High-level view of the rich pipeline of tools used in RACK.

At the bottom is a SPARQL 1.1 compliant triplestore. For practical and programmatic reasons, we use Apache Fuseki [10] for this stage of the program. Over that is the Semantics Toolkit (SemTK) service layer [11][12]. A suite of semantics tools that has been in development at GE for several years, SemTK is based upon the concept of "nodegroups" [13] which are graphical depictions of a subgraph of interest, with additional annotations for automatic generation of queries including ingestion. SemTK includes the SPARQLgraph interface for visually editing nodegroups and ingestion mapping templates, running queries, and performing various utility functions. It also provides Java and REST APIs for asynchronous queries, ingestion, and utility functions.

Then there is a Python API [14] upon which several other tools are built, including a user interface, RACK UI, which supports basic data loading and links to SPARQLgraph, and a command-line interface.

The manifest format defines the large and complex data packages that are used to populate RACK. It is comprised of sub-folders of OWL ontology files, CSV data files, nodegroup queries, ingestion templates, SemTK reports, and YAML files that describe the manifest's components and loading sequence.

As described in the previous section, SADL is used to compose the ontology, and the SADL tools compile into OWL, which is then loaded via the manifest. Other important tools include the Scraping Toolkit, which accesses the ontology through the semtk-python layer and is then used to build document scrapers which output CSV and YAML files for the manifest. The Entity Resolution Tool uses semtk-python to access both the ontology and previously loaded data to create CSV files that describe entities which can be combined in an additional, cleaner copy of the data.

Curation of data from multiple sources involves the orchestration of many complex processes. The pipeline introduced above allows us to load large ingestion packages successfully and repeatably. RACK can ingest a data package of 2.5M triples in under 15 minutes, inclusive of type checking, URI lookup and linking, and verification of the domains and ranges of all properties. The APIs allow other ARCOS performer teams to build and integrate RACK features and capabilities into their own tools. The following sections of this paper will describe key components of RACK in more details.

4.1 SemTK Nodegroups

The "nodegroup" is a fundamental concept of the SemTK layer upon which RACK's data ingestion and querying is built. A simple example is shown in **Fig. 6**. The nodegroup is edited using SPARQLgraph by dragging classes from the ontology onto the canvas and selecting desired properties. It represents a subgraph of interest for a particular query or queries. Pathfinding assists the editing process and may be based either solely upon the ontology or account for existing instance data when choosing likely paths. By default, a node represents not only the class shown, but all subclasses. Data properties appear by name only, and object properties as links to additional nodes. The graphical editor provides a rich set of annotations that allow properties to be selected for return, labeled optional or minus, filtered, and have functions applied. Object properties may have operators such as * and + applied. Unions are supported. Although a very useful subset of SPARQL is supported, nodegroups have yet to support the full expressivity of SPARQL.

The right side of **Fig. 6** shows SPARQL for a "select distinct" query that is autogenerated by SemTK, with some prefixes removed and edits made for readability. Query types such as count, construct, ask, and delete are all supported. Nodegroups are stored in a "nodegroup store" and referenced by id. When applied to a different SPARQL connection, "FROM" or "USING" clauses are updated appropriately. Stored nodegroups support the concept of "runtime constraint," which allow the APIs to assist in sending along constraints such as *?requirement_id = "Req1"* when a query is executed. It should be noted that since the SPARQLgraph editor is ontology-driven, all

SPARQL-generated queries (and more importantly, the ingestion process described below) are consistent with the ontology.

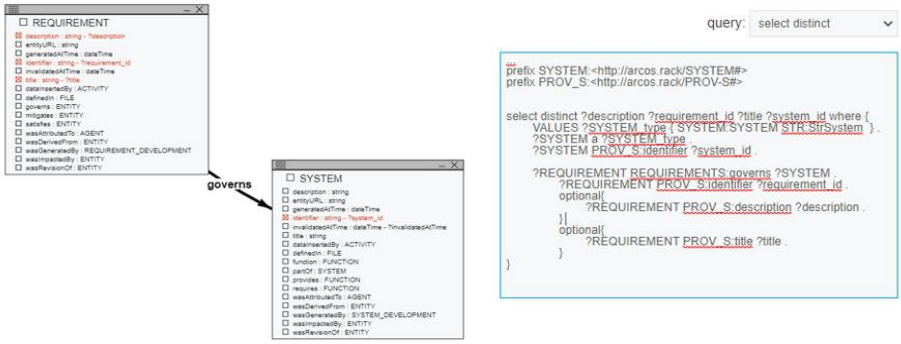


Fig. 6. Simple nodegroup and auto-generated SPARQL

4.2 SemTK Ingestion Templates

Building upon the nodegroup concept are ingestion templates. A template consistent with the nodegroup in **Fig. 6** is shown in **Fig. 7**. A template maps the data in each line of a CSV file to a nodegroup such that the triples representing one nodegroup subgraph will be inserted for each line of the CSV.

The right side of the template shows the names of four columns in a target CSV file. The left shows where each maps to the nodegroup. With certain exceptions related to lookup and CSV validation features, empty or missing values are pruned from the nodegroup before the line of CSV data is inserted.

The ingestion process supports many features beyond the scope of this paper. However, a few of them are critical to the RACK ingestion process. Most importantly is the *URI Lookup* feature. Note that all the bolded rows in the ingestion template (which represent classes) are set to "—Generate UUID," meaning that random URI's will be generated during ingestion and not built from strings in the CSV. The darkened boxes near *?SYSTEM* and *?system_id* indicate the instance of the class SYSTEM will be looked up in the target graph using the property *identifier* (the property with object *?system_id*) and the value from the column *system_id*.

When looking up URIs, the template provides the options of *create-if-missing*, *error-if-missing*, and *error-if-exists*. For the first two more commonly used options, once the URI is found, other properties will be added to the existing data. RACK uses these features to store all data with random URIs and avoid the need to publish and enforce standards for URI construction across different program performers. Instead, data is looked up by identifier.

In addition to *error-if-missing*, the ingestion process also checks the format of all data presented as URIs, dates, times, numbers, etc. and prevents ingestion of any data that is not properly formatted. This works alongside the ontology-driven nodegroup used to ingest each row, which guarantees that the types, properties, domains, and

ranges in all the triples conform to the ontology. These represent RACK's first level of data verification on incoming data before it enters the triplestore.

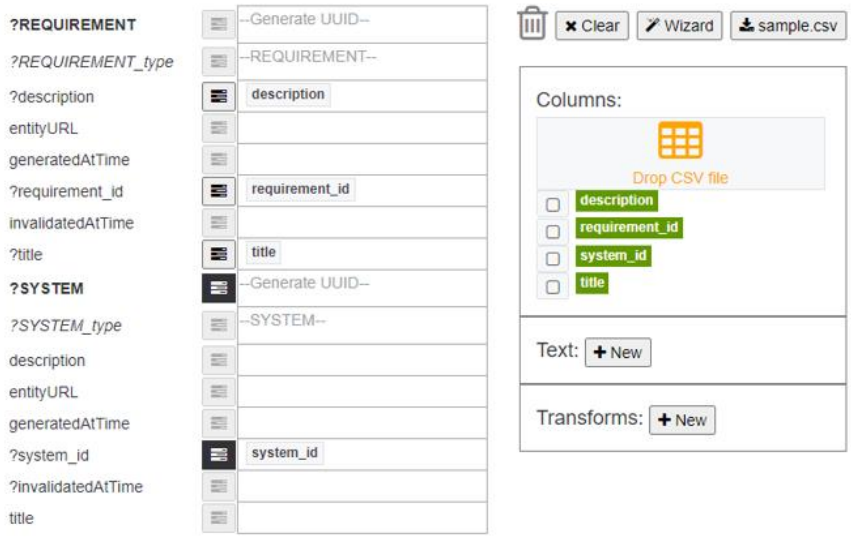


Fig. 7. An ingestion template.

4.3 Automatic Ingestion Templates ("Ingest by Class")

The SemTK nodegroup and template-based ingestion process described above is both powerful and very flexible. We determined that a less flexible standardized ingestion template would speed understanding and adoption of ingestion across the many performers on the program. To this end, a standard ingestion template can be automatically generated for any given class.

RACK's auto-generated class ingestion templates have the following attributes. There is one instance of the target class that is looked up by *?identifier* in the mode *create-if-missing*. This means that an instance of the target class will be created if needed, but if one already matches the *?identifier*, then the data from the row will be added to the existing instance. Each data property has a corresponding column in the CSV. Each outgoing object property's object will also be looked up by its identifier, using a column name of the format: *<propertyShortName>_identifier*. The objects of object properties are looked up in the mode *error-if-missing*. This has the positive effect of preventing ingestion of links to non-existent instances. It does, however, force data to be ingested in a specific order such that object property objects always exist. In cases where circularities exist, custom templates must be used instead of these auto-generated versions.

RACK provides additional features to ease the use of auto-generated templates. There are API calls to get a copy of the nodegroup for inspection, and to get sample CSV files compatible with the template. Nodegroups have all properties set to optional

so they may also be used to generate SELECT or CONSTRUCT queries to explore ingested data.

Despite the loss of expressivity, the forced ordering of loading, and in many cases loss of performance compared to customized ingestion templates, these auto-generated templates have proven very easy to understand and use. Most of the data ingestion in RACK to date has leveraged this feature.

3.3.1 Checking Quantified Cardinality.

SemTK has a copy of the ontology including cardinality constraints available to it during ingestion, but it is not practical to enforce quantified cardinality during the ingestion process. Actual counts will be lower than cardinality limits while ingestion is incomplete. With multiple-step ingestion packages, it is not possible to report such errors during a single ingestion. Actual counts exceeding cardinality limits would be possible but inefficient.

We chose instead to provide a SemTK REST function for checking quantified cardinality that can be run after ingestion of the entire dataset is complete. This is available in the reports feature, described in a later section, and accessible through various APIs. The cardinality check reads through the ontology and runs a SPARQL query for each cardinality restriction. It returns a table of violations consisting of the following fields:

- class – the class with the restricted property
- property – the property being restricted
- restriction – type of restriction (max, min, qualified)
- limit – cardinality limit declared in the model and violated in the data
- subject – the instance of the class that violates the restriction
- actual_cardinality – the actual cardinality of the property for this subject

4.4 Ingestion Packages

To maintain a clean separation of data curation and data exploration, and to conform to the data owner's specification that data be stored locally, RACK supports an archive format that contains a complete copy of the ontology, datasets, queries, and reports. These "ingestion packages" can be generated using our data curation tools and ensure that the resulting RACK state is easy to reproduce for multiple users and across multiple reloads.

This reproducibility has been important for enabling us to make sweeping changes as the ontology and dataset evolved without sacrificing the ability for users to work with past versions of the data. Each ingestion package comes complete with the appropriate version of the ontology. Internally the ingestion packages are structured as zip files containing the various component OWL, CSV, and JSON files indexed by YAML files. Manifest files indicate how the content should be loaded into RACK. Model and data manifest files describe the sequence in which OWL or CSV files are loaded, which triplestore graphs to load them to, and which ingestion templates to use for each. Manifests may also specify nodegroups (for queries or custom ingestion templates) to be loaded to the nodegroup store. Beyond the data and ontology, an ingestion package

contains enough information to indicate target graphs, clear data, perform entity resolution and other maintenance tasks performed at load-time.

Significant support has been built to facilitate the use of ingestion packages within our program. The RACK command line interface[3] supports automatically building ingestion package zip files given a set of manifest files. Ingestion packages may be loaded into RACK using the command line interface or a simple RACK user interface page (highlighted later in section 6). Further, as zip files, ingestion packages may be easily shared between users and reliably reproduced into any given RACK installation. Notably, these operations require no semantic expertise and therefore lower the barrier to entry for using RACK.

4.5 Scraping Toolkit

The norm for system development is for a multitude of tools and methodologies to be used, each creating evidence supporting certification. Generically, data curation must extract raw data from source material, perform data transformation on that raw data to define it per the ontology, and format it as an ingestion package. One of the support tools provided with RACK is the Scraping Toolkit (STK)[4] that facilitates the collection of evidence into ingestible packages. Fundamentally, the STK is a Python library that is auto-generated from the ontology. The STK provides methods for collecting the evidence that are automatically curated into fully formed ingestion packages from raw evidence using the ingestion templates.

Raw sources of evidence are processed by creating a Python script that identifies the individual pieces of evidence. Scripts start with "CreateEvidenceFile", a method that initializes a RACK-DATA.xml file that is used to collect bits of evidence as they are found while processing the source material. When evidence is found, there are "Add" methods that append evidence to the RACK-DATA.xml file. Each data class has its own "Add" method that is customized to its ontological definition, with optional parameters for each property. As additional evidence is found while processing the source material, the RACK-DATA.xml is continually expanded until the end of the script at which time the "createCDR" method processes the RACK-DATA.xml to create a fully formed ingestion package. This process simplifies the creation of the transformation scripts by not requiring the data provider to have to sort and manage interspersed data, but rather can just record all information that is found as it comes while processing the source material—the processing in "createCDR" curates and combines evidence.

Python works well as it is an easy, well known programming language and there is a large library of existing modules that can be used. Regular Expressions in Python are easy to use and libraries for XML, DOCX, and even source code exist that can be combined with the STK to make data extraction from nearly any source relatively simple.

---

[3] https://github.com/ge-high-assurance/RACK/tree/master/cli

[4] https://github.com/ge-high-assurance/RACK/tree/master/ScrapingToolKit

4.6 Entity Resolution

Any time data is collected from multiple sources there's a possibility of creating different instances within the same data graph even though it's meant to be the same piece of evidence. Small inconsistencies that a human might miss are inconsistent capitalization or the use of different types of dashes. While it is generally preferable to have all these disconnects addressed as part of the scraping process this is not always possible.

To accommodate this inevitability within RACK, we perform entity resolution to resolve these issues without modifying the source data graphs. In the RACK ontology, a SAME_AS class is defined with two object properties: *primary* and *secondary*. When a user identifies instances within the data graph that should be combined, a new SAME_AS instance is created that connects the two instances. To aid in this process, we created an Entity Resolution Tool that evaluates the likelihood of possible matches based on a set of rules guided by the ontology. For example, two instances that have the same identifier, and one's type is a super class of the other would have a high likelihood representing the same evidence, as would two instances with the same class with identifiers that only differ by a hyphen. Alternatively, two instances that represent evidence from the same source material with nearly identical identifiers (for example, SRS_Req*1*, SRS_Req*2*) will have a lower probability of representing the same evidence. These rules can be customized, and the Entity Resolution Tool will extract possible matches from the data based on class compatibility, score them based on the defined rules, and present the results to a user for final determination.

While this creates a relationship within the data graph, usage of this relationship in queries would be complicated as now every node within the search would have to allow for this potential. Rather than leave this complication to the user to deal with, we collapse these SAME_AS relationships into a single, additional resolved data graph. This cleaner copy of the data graph has all the instance data from the source graphs except any object that was identified as a secondary is merged into the primary. Any conflicts related to this merge is resolved by using the primary's data. This new graph allows queries to be run that do not have to account for the SAME_AS complexities while leaving the original source graphs untouched and available for review. Maintaining the source graph is still necessary for providing providence of the evidence—from source material to the final evidence graph a clear line can still be drawn.

5 Evidence Exploration

Our vision for RACK is that data exploration will be accomplished with tools built by users on top of the RACK APIs. We have built query nodegroups that serve both as stand-alone data verification tools, and as samples for data retrieval that is performed by such tools. Further, RACK includes a generic reporting tool that allows queries to be organized into sets that are of interest to a particular role, and repeatably executed in bulk into a summary page.

5.1 Queries

The RACK tool includes over 30 pre-defined queries[5] in the form of nodegroups that are pre-loaded into the SemTK nodegroup store during ingestion of manifests.

A dozen of these queries are meant as follow-ups to cardinality checking. They query instances of and give additional information about data that has violated a particular cardinality constraint such as a REQUIREMENT without a *description*. This additional information proves useful in troubleshooting ingestion problems. But other queries that check constraints are slightly more complex than can be easily expressed in SADL. In one example, the REQUIREMENT was intended as an abstract superclass, so a query is provided to check for instances of this class. Pre-defined data verification queries can check for OR conditions, such as finding requirements that do not either satisfy another requirement or govern a system component.

Other queries show structure of the evidence. They take a runtime constraint of an entity such as a REQUIREMENT, SYSTEM, or HAZARD and produce tree or table output showing how requirements or systems roll up to each other. In the case of HAZARDs, the sample query shows the entity from which the hazard was derived, and the requirement tree that attempts to mitigate it, if any.

Finally, there are queries that demonstrate how to check the evidence in an assurance case. Examples include finding requirements with no passed tests, or those specifically with failures, or those with no tests at all. These query nodegroups are intended to be useful as well as demonstrative of how users can write their own as they manually explore the data or build tools on top of RACK APIs.

5.2 Reports

Nodegroups can be assembled—along with a few special functions—into reports, which can be stored as JSON files and drag-and-dropped into SPARQLgraph, or in the nodegroup store and accessed by id. A simple editor using *jsoneditor* [15] inside of SPARQLgraph allows users to assemble the queries that impact their specific role (i.e., was the data loaded properly vs. is the evidence complete vs. are all systems governed by REQUIREMENTs with passing TESTs) and re-run the group of queries in a single step.

The “special functions” include the cardinality checker described earlier, and a simple count of the number of instances of each class. Each nodegroup may be executed as SELECT DISTINCT to tables, or as CONSTRUCT queries to network graphs. For tables, success or failure may be defined with simple row count constraints. So, a query for bad data (e.g., an INTERFACE without a source) that returns no rows simply shows up as *success,* whereas one that finds rows is labelled *failure* and the table of results included.

Although not intended to replace evidence-checking applications that may be built on RACK, the report provides a powerful ability to run a large number of queries in

[5] https://github.com/ge-high-assurance/RACK/tree/master/RACK-Ontology/nodegroups

one step and produce pages of output which are easy to scan for issues, and which provide some level of interaction (such as sorting and filtering tables).

6 Impact and Lessons Learned

RACK has broad adaptation in the ARCOS research community, including teams at Lockheed Martin, SRI, GrammaTech, STR, RTX, Honeywell, and more. The communal ontology allows data providers (TA1 teams) to ingest their evidence, and in some cases, to extend the ontology with subclasses and sub-properties germane to their tools.

Lockheed Martin's tool called CertGATE produces Evidential Assurance Case Fragments (EACFs), which are structured arguments linked to pieces of evidence that are ingested into RACK. Similarly, SRI's DesCert team created an extensive ontology extension[6] to capture the evidence produced by their multitude of tools [16]. The evidence "interface" in RACK enabled software and documentation analysis tools, like the one developed by the A-CERT[7] team lead by GrammaTech, to uniformly capture evidential claims: claim that a certain property holds about the system, link the claim to the relevant design and implementation elements, and supply raw evidence supporting the claim. The assurance case research teams (TA3s) have also broadly integrated RACK into their tools. STR's ARBITER currently automates the harvesting of data from RACK, organizes it as guided by assurance reasoning strategies, and presents a candidate assurance case for review by an end user via a browser-based GUI. Similarly, RTX's AACE, an Automatic Assurance Case Framework, instantiates security case patterns using data provided by its evidence manager, which is informed by RACK [17][18]. Honeywell's Clarissa tool maps RACK evidence into logic programming, which led to research breakthroughs such as target constrained natural query language built on top of the RACK ontology within a principled, structured case adhering to Assurance 2.0 methodology [19].

One lesson learned in working on the ARCOS program and serving as the evidence curator and ontology maintainer is that users need help with triplification. Non-semantic experts need help getting data into the triplestore. Requesting RACK users provide their certification evidence in the form of triples would have required extensive training and subsequent debugging and may have jeopardized the approach. So, we introduced the performer teams to SemTK and default class ingestion templates, which use the ontology to map CSV files to triples while performing significant verification. To make the tool programmatically interactive, we provided Java and REST APIs, and we also provided Python-based command line interfaces. These interfaces helped, but still some users struggled with installation. The game changer was a Dash[8]-based RACK UI web page that lessened the friction for end users to load data into RACK (**Fig. 8**). By turning data ingestion into a 1-click process, data providers can focus on delivering high quality data instead of agonizing over multi-step installation. Data consumers indirectly reap these benefits because the easier it is to ingest high quality data into RACK, the bigger

[6] https://github.com/ge-high-assurance/RACK/blob/master/overlays/SRI-Ontology/ontology
[7] https://grammatech.github.io/prj/acert/
[8] https://dash.plotly.com/

the treasure trove of information to mine to show certification standards compliance. The user simply zips up the ontology files and data files into an ingestion package, along with the manifest that lists the underlying model footprint and load steps. On ARCOS, the target DoD system contained approximately 170 CSV data files. We are also developing an integrated environment called RITE (**R**ACK **I**ntegrated cer**T**ification **E**nvironment)[9], which integrates SADL and RACK. This will allow users to write or modify an ontology and quickly prove its usability via sample data. It will also assist users in composing ingestion packages, which as noted, can contain hundreds of data files.

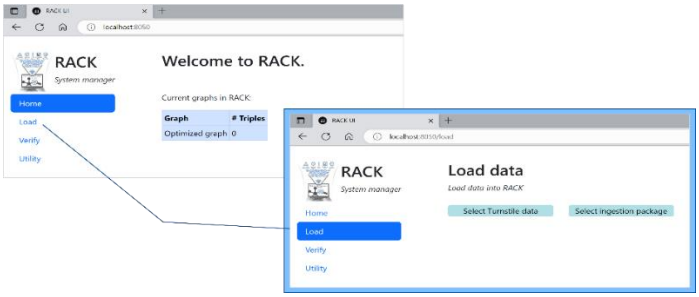


Fig. 8. RACK UI page with 1-click data load and 1-click data verification.

7 Conclusions

In this paper we described how we were able to bring the Semantic Web stack into the aviation certification domain. The graph view and recursive graph query operators in SPARQL are an intuitive fit to evidence curation. The ability to write an ontology with natural support for subclasses, sub-properties, and cardinality constraints, and to write inferences to ensure model compliance were a natural fit. Our most recent ingestion package consists of approximately 2.5M triples. RACK has been integrated into the tools of at least six other performer teams.

Acknowledgement and Disclaimer. Distribution Statement A. Approved for public release: distribution unlimited. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government

[9] https://github.com/ge-high-assurance/RITE

References

1. ARCOS Homepage, https://www.darpa.mil/program/automated-rapid-certification-of-software.html, last accessed 2023/04/14.
2. psac-template, https://studylib.net/doc/26041621/psac-template, last accessed 2023/07/28.
3. Overview of Rational® DOORS®, https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.6.0?topic=overview-rational-doors, last accessed 2023/07/28.
4. McMillan, C., Crapo, A., Durling, M., Li, M. et al., "Increasing Development Assurance for System and Software Development with Validation and Verification Using ASSERT™," SAE Technical Paper 2019-01-1370, 2019, doi:10.4271/2019-01-1370.
5. Crapo, A. and Moitra, A.: Toward a unified English-like representation of semantic. In: International Journal of Semantic Computing, vol. 7, no. 3, pp. 215-236 (2013).
6. SADL GitHub Page, https://github.com/SemanticApplicationDesignLanguage/sadl, last accessed 2023/04/14.
7. OWL Homepage, https://www.w3.org/OWL, last accessed 2023/04/14.
8. Xtext Homepage, http://www.eclipse.org/Xtext/, last accessed 2023/04/14.
9. Moitra, A., Cuddihy, P., Siu, K., Archer, D., Mertens, E., Russell, D., Meng, B., Interrante, J., Quick, K., Robert, V.: A semantic reference model for capturing system development and evaluation. In: International Conference on Semantic Computing (ICSC), Laguna Hills (2022).
10. Fuseki Homepage, https://jena.apache.org/documentation/fuseki2, last accessed 2023/4/18.
11. Cuddihy, P., McHugh, J., Williams, J.W., Mulwad, V., Aggour, K.S.: SemTK: A semantics toolkit for user-friendly SPARQL generation and semantic data management. In: 17th International Semantic Web Conference (ISWC), Industry and Blue Sky Ideas Track, Monterey, CA, (2018).
12. Semantics Toolkit GitHub age, https://github.com/ge-semtk/semtk, last accessed 2023/4/18.
13. Kumar, V.S., Cuddihy, P., Aggour, K.S.: 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, pp. 1-4. (2019).
14. Semtk-python GitHub Page, https://github.com/ge-semtk/semtk-python3, last accessed 2023/4/18.
15. Jsoneditor GitHub Page, https://github.com/josdejong/jsoneditor, last accessed 2023/4/13.
16. Shankar, N., Bhatt, D., Ernst, M., Kim, M., Varadarajan, S., Millstein, S., Navas, J., Biatek, J., Sanchez, H., Murugesan, A. and Ren, H.: DesCert: Design for Certification. arXiv preprint arXiv:2203.15178 (2022).
17. Wang, T. E., Daw, Z., Nuzzo, P., & Pinto, A.: Hierarchical contract-based synthesis for assurance cases. In: Proceedings of NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, pp. 175-192. (2022).
18. Oh, C., Naik, N., Daw, Z., Wang, T. E., & Nuzzo, P: ARACHNE: Automated validation of assurance cases with stochastic contract networks. In: Proceedings of Computer Safety, Reliability, and Security: 41st International Conference (SAFECOMP 2022), Munich, Germany, pp. 65-81. (2022).
19. Bloomfield, R., Rushby, J.: Assurance 2.0: a manifesto. In:arXiv preprint arXiv:2004.10474 (2020).