Australian Government
Department of Defence
Defence Science and
Technology Organisation

Testing the Shapes Vector derived UniSec Ontology in OWL using Protégé and Pellet

Tamas Abraham and Dean Philp

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation

DSTO-TN-1052

ABSTRACT

This document describes the details of implementing the UniSec ontology in the Web Ontology Language (OWL) using the Protégé ontology development tool, and testing it with the Pellet reasoning engine to evaluate its usefulness.

RELEASE LIMITATION

Approved for Public Release

.

Published by

Command, Control, Communications and Intelligence Division
DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

Telephone: (08) 7389 5555
Fax: (08) 7389 6567

AR-015-172
December 2011

APPROVED FOR PUBLIC RELEASE

Testing the Shapes Vector derived UniSec Ontology in OWL using Protégé and Pellet

Executive Summary

The Shapes Vector project is well into its second decade. The framework uses an ontology design that, while state-of-the-art at the time, does not feature some concepts that have been developed in various research communities since its inception. It is also proprietary in its implementation, which means that it cannot make direct use of many recent, readily available tools that could supplement its capabilities and contribute to its wider adoption.

This project investigates whether a current ontology standard, such as OWL, and the many tools developed for its use, can provide Shapes Vector with further reasoning capability. Such ability is meant to be provided in an "add-on" fashion: the Shapes Vector framework will remain the processing workhorse for input handling and much of the reasoning, except in those cases where it does not provide the necessary capability. In those instances, the existing ontology will be exported to OWL, reasoned with, then imported back into SV for further processing, if necessary.

This page is intentionally blank

Contents

This page is intentionally blank

1. Introduction

This document describes the details of implementing the UniSec ontology [1] in the Web Ontology Language (OWL) using the Protégé ontology development tool, and testing it with the Pellet reasoning engine to evaluate its usefulness.

Some knowledge of OWL is necessary to fully understand the process we describe. However, the main source of documentation [2] we used should be sufficient for most readers.

Throughout this document, we shall highlight the differences we have found between our OWL-based solution and the original Shapes Vector (SV) Cyber1 ontology [3]. In this respect, this document can serve as a guide to those wishing to do their own conversion of an SV style ontology to OWL. One major omission is the treatment of set definitions, which are defined in Cyber1 but are not part of the UniSec ontology.

2. Experiment Purpose Statement

The Shapes Vector project is well into its second decade. The framework uses an ontology design that, while state-of-the-art at the time, does not feature some concepts that have been developed in various research communities since its inception. It is also proprietary in its implementation, which means that it cannot make direct use of many recent, readily available tools that could supplement its capabilities and contribute to its wider adoption.

This project investigates whether a current ontology standard, such as OWL, and the many tools developed for its use, can provide Shapes Vector with further reasoning capability. Such ability is meant to be provided in an "add-on" fashion: the Shapes Vector framework will remain the processing workhorse for input handling and much of the reasoning, except in those cases where it does not provide the necessary capability. In those instances, the existing ontology will be exported to OWL, reasoned with, then imported back into SV for further processing, if necessary. A more detailed description of the ideas behind this can be found in [5] and [6].

3. Ontology Implementation

This section details the conversion mechanism from the Shapes Vector Cyber1 ontology to OWL. Such conversions (and their inverse) can be automated in SV by the use of *bridges* (the discussion of which not being part of this paper).

3.1 Taxonomy

The class hierarchy representation in OWL is derived from a root object called Thing. Otherwise, creating *Classes* (called *Entities* in Cyber1) is a straightforward process using

the Protégé interface and the resulting taxonomy should mirror exactly the entity structure we would build for SV.

3.2 Relationships

Relationships between entities in Cyber1 are represented by *Object Properties* in OWL. An Object Property is just a definition that, unlike Cyber1 relationships, can have some defined Characteristic, such as being Transitive, Functional, Symmetric, and so on. In addition, an Object Property has its own Descriptions, most notably its Domain, Range and Inverse. The latter allows the inverse of an Object Property to be directly defined, while in SV, this has to be supported by the implementation. The other two Descriptions, Domain and Range, are used to associate two or more Classes via the Object Property.

In practice, however, the Protégé manual recommends not defining a Domain or Range for Object Properties (with one exception, see later in this document). It is better to create a relationship between one Class and another through an anonymous *Superclass* that will directly link the two Classes. For example, in the Class definition for Display, one could add a Superclass '**displays** *some* Content', which links Display to Content via the Object Property **displays** (and specifying the *some Existential Restriction*, i.e. an 'at least one' relationship). This allows the Object Property to be re-used in multiple relationships without explicitly defining the Classes involved as Domains and Ranges. For example, if we wanted to use **displays** in another context such as a Person displaying Emotion, using Domains and Ranges, we would specify both Display and Person as Domains, and Content and Emotion as Ranges. This may, however, erroneously imply that a Display can also display Emotion, which may not have been an intended relationship within the ontology.

All relationships from the Cyber1 ontology can be translated to OWL Object Properties in this manner without difficulty.

3.3 Attributes

Entity *Attributes* in Cyber1 are treated in two different ways in OWL. This depends on whether a particular attribute is represented as a primitive variable, such as a number, string, or binary, or by a variable that takes a value from a limited number of pre-defined choices (a set of non-primitive values).

In our definition of the UniSec Ontology, we have simplified the multitude of different primitive data types available in Cyber1 down to a few, such as string, integer and hexBinary. Attributes taking such values are represented by *Data Properties* in OWL.

Data Properties are associated with Classes in a fashion similar to Object Properties: as anonymous Superclasses of individual Classes via an Existential Restriction. For example, the name of a Computer can be added as a Superclass to the definition of *Computer* as '**name** *some* string', where **name** is a Data Property defined elsewhere and string is a variable type which is part of OWL. Data Properties, just like Object Properties, have the Domain and Range Descriptions, and a single Characteristic, Functional. Again, the Protégé manual advises against defining Domains and Ranges for Data Properties. On the other hand, OWL has the flexibility to constrain the range of values when defining an anonymous Superclass: if we, for example, wanted to limit the **size** of a Display to a maximum of 100 inches, we could simply say '**size** *some* positiveInteger[<=100]'.

The Functional Characteristic is an OWL construct that says that a Data or Object Property links Classes with an 'only one of' restriction. That is, an individual from a given Domain can only link to a single individual in the Range. An interesting side effect of this restriction is that if an individual links to two separate individuals in the Range, then the two targets will be assumed to be the same individual.

This Characteristic is utilised in the other treatment of Cyber1 attributes, those that can take values from a small set of possible choices. For example, the type of a sensor may be limited to a motion sensor, RFID reader, iris scanner and so on. These attributes are represented in OWL by Object Properties in a special way.

First, every value that is available to these attributes will be defined as Classes under a special ValuePartition class, which itself is a subclass of Thing. Each type of attribute will be a Subclass of ValuePartition. Individual values for each attribute type will be Subclasses of the type Superclass. For example, the Class SensorType (a Subclass of ValuePartition) will have its own Subclasses MotionSensor, RFIDReader, IrisScanner and so on. SensorType can then, for example, be used for the **type** attribute in the Class Sensor. The Subclasses of SensorType are made *disjoint* to state that each Subclass is distinct. A *covering axiom* for SensorType is also provided to ensure that the listed Subclasses are the *only* values that can be taken by an attribute that takes a SensorType value, such as **type** in Sensor. Finally, SensorType is made *Functional*, ensuring that only one of the Subclasses is associated with an individual of a Class that has a SensorType attribute.

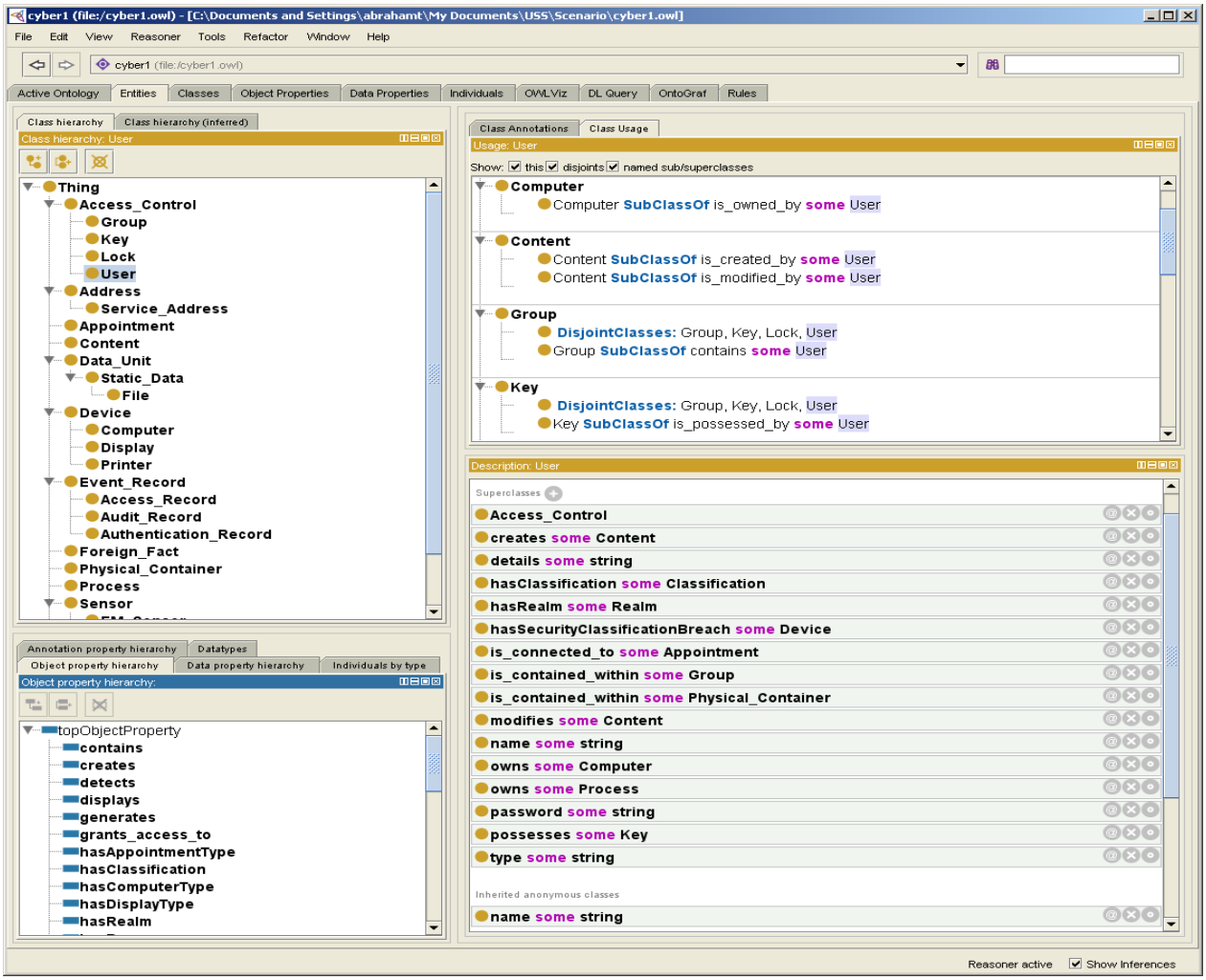

Figure 1 Protégé screenshot showing Entities and Properties.

With these two approaches (Data Properties and ValuePartition Subclasses), attributes in the Cyber1 ontology can be effectively mapped into OWL representations.

Figure 1 shows part of the UniSec ontology, with the taxonomy displayed in the top left corner, some of the defined relationships in the bottom left corner. The Class User has been highlighted, with the bottom right corner window showing some of its attributes, and the top right corner displaying some relationships it has with other classes.

3.4 Other Aspects

In this section, we list some comments on the conversion process that may prove helpful for future users.

OWL requires Classes to be specified disjoint in order to ensure that an individual is a member of only one class. This is not the default behaviour in Protégé and must be enforced manually when creating the class hierarchy.

Some Entities in Shapes Vector are defined as Virtual. Such classes disallow instantiation. No such restriction exists in OWL, and care must be taken when creating individuals so that they are members of a class that corresponds to a concrete Entity in SV.

The covering axiom that the manual suggests for ValuePartition Classes shows that these Classes are *defined*. That is, the axiom is saved as an *Equivalent Class* definition. However, when using Protégé's built-in covering axiom generator, this definition is created as an *anonymous Superclass* instead. We have followed the instructions from the manual and moved the auto-generated covering axioms from Superclass to Equivalent Class.

4. Scenario Descriptions

Once the ontology is defined, the next step is to create scenarios that we are interested in observing. These may include policy violations, impossible events (such as being in two places at once), and so on. The idea is to then use a reasoner that will investigate our ontology and report occurrences of these scenarios to us.

The practical requirements of working with these scenarios include the ability to dynamically adjust them. This way, a developer can write agents that adapt to different applications, and can perform targeted incident detection with on-the-fly modifications. It also provides the user with control over agent resource consumption (e.g. CPU), and lets operators load background knowledge for reasoning as required.

Scenarios should also be able to be expressed as goals supported by one or more conditions, with each scenario detected only once for each matching set of individuals.

In this evaluation, we use SWRL [7] as the means to set scenario goals and conditions in our OWL ontology. In the current version of Protégé we cannot create a new individual using pure SWRL rules, until the SWRLX Ontology [8] gets ported to it from version 3.X (which contains a makeOWLThing construct) [9]. There are other ways for creating individuals using rules in OWL [10], but these are not accepted as Description Logic (DL) Safe and we have therefore discounted their use. Alternatively, we could have considered the use of the SPARQL 1.1 Query Language [11]. However, we prefer SWRL over SPARQL because its rules are declarative, which are usually easier to understand. In addition, CLIPS, the tool Shapes Vector uses for reasoning, has declarative-style rules, and thus the learning curve for existing users will likely be smaller.

We can thus represent our scenarios [12] as relationships between two classes (via anonymous superclasses) and the definition of an SWRL rule. An example relationship, hasSecurityClassificationBreach, can be added using the following procedure:

- Add Object Property hasSecurityClassificationBreach
- Add hasSecurityClassificationBreach as inverse property of itself
- Add Superclass to User as hasSecurityClassificationBreach some Device

- Add Superclass to Computer as hasSecurityClassificationBreach some Device
- Add individual **RESTRICTED** Types Restricted
- Add individual **SECRET** Types Secret
- Define SWRL rule to state that if a User is only **RESTRICTED**, and a Computer is **SECRET**, then if an individual of the User class owns such a Computer individual, then this will represent a breach relationship:
    - *Computer(?c), User(?u), hasClassification(?c,SECRET), hasClassification(?u, RESTRICTED) , owns(?u,?c) -> hasSecurityClassificaitonBreach(?c, ?u)*

Here, User and Computer are concrete Entities from Shapes Vector, transferred into our OWL representation. Device is the virtual parent Entity of Computer. Restricted and Secret are Subclasses of Classification, a ValuePartition Subclass created in OWL to represent attribute values (see Section 2.3). Note that explicit individuals of these attribute classes need to be created for SWRL to work. This is just a one-to-one mapping of an attribute class and its only representative individual (created using all capital letters for easier reading).

To test the above rule, we created two additional individuals with some relationships. We added the individual **Computer1** Types Computer, with Object Property assertion hasSecurityClassification **SECRET**. We also added the individual **dmp** Types User, with Object property assertions hasClassification **RESTRICTED** and owns **Computer1**, which clearly constitutes a security breach as per our above definition. This breach, however, was not asserted into the ontology as our aim is to make sure that we are able to infer this knowledge with a reasoner. Checking the now rule-extended OWL ontology with an appropriate reasoner is easily done inside Protégé using plugins or the built-in reasoning engine. We chose the Pellet plugin [13], which did indeed make the correct inference **dmp** hasSecurityClassificationBreach **Computer1** when started. For the sake of completeness, we mention that two additional inferences were also made by Pellet, the inverse of the above, **Computer1** hasSecurityClassificationBreach **dmp**, as well as the inverse ownership relationship, **Computer1** is_owned_by **dmp**.

We can also use DL Query to manually confirm the scenario goal:

- In Protégé, the "Individuals" check box ensures desired results
- Query 1: "User and hasClassification some Restricted and owns some (Computer and hasClassification some Secret)"
- Query 2" Computer and hasSecurityClassificationBreach some User"

5. Standalone Reasoning

5.1 OWL APIs

For the purposes of testing, we chose the Java language due to its prevalent use within the semantic Web research community. Several different free APIs are available for implementing standalone applications in Java that are capable of representing an ontology as defined above:

- The Jena API, which was originally developed for the Resource Description Framework (RDF) but now contains OWL specific functionality.
- The OWL API, used by Protégé v4 among others, which started as the Wonderweb project (version 1), continued as CO-ODE and TONES (version 2), and is currently maintained at the University of Manchester (version 3)
- The Protégé OWL API, used by Protégé version 3

Reasoning with our ontology can be done with a multitude of free semantic reasoners. They include

- HermiT, a calculus based reasoner,
- FACT++, a Description Logic classifier, and
- Pellet, another DL reasoner that comes with a Java API.

A large number of other reasoners and OWL APIs are also available, see for example [4] for a more complete list.

For maturity reasons, we chose Pellet for our test, and coupled it with the version of OWL that comes with the latest iteration of Protégé, OWL API 3 from the University of Manchester. The current version of Pellet is being maintained by Clark & Parsia, and has evolved from a version originally created by the Mindswap semantic web research group at the University of Maryland.

5.2 Implementation Details

What follows below is the implementation path we have used to verify that it is indeed possible to use and modify our OWL ontology programmatically, while generating and recognising inferences using an attached reasoner. We do not claim that this implementation is optimal and should be followed when working on a similar project. There may be simpler, more elegant and efficient solutions to the problem described below, and a variation to the problem definition may require a completely different approach to produce results. However, some of the solution presented will be universally required (e.g. how to access the ontology or run the reasoner).

The process we went through consisted of only a few important steps. We wanted to:

1. Access our saved ontology
2. Attach a reasoner and do an initial evaluation of the existing ontology
3. Add new ontology elements (individuals and relationships)
4. Incrementally infer changes and report them

5.3 Code Samples

This section contains Java code snippets with explanations in a sequential order as one would find in a program. We start with the ontology details.

```
/* Define Ontology */
```

```
String ontolocation = "http://locationofOWLontology/cyber1.owl";
IRI ontologyIRI = IRI.create(ontolocation);
String physicallocation = "file:/somewhereondisk/cyber1.owl";
IRI documentIRI = IRI.create(physicallocation);
SimpleIRIMapper mapper = new SimpleIRIMapper(ontologyIRI, documentIRI);
```

We continue with the loading of the ontology.

```
/* The three steps of loading and making an ontology accessible */
OWLOntologyManager manager = OWLManager.createOWLOntologyManager();
OWLDataFactory factory = manager.getOWLDataFactory();
OWLOntology ontology = manager.loadOntology(ontologyIRI);
```

Add a reasoner and perform initial inference computations. If our ontology was correct, there should be no inferences made now, since we only have class definitions, relationships and SWRL rules. Note that in our discussions in the previous section, we mentioned that we did add some individuals to our ontology already using Protégé, so we expect the corresponding breach to be found here.

```
/* Use a non-buffering reasoner (a buffering reasoner would not */
/* process any additions until manually refreshed) */
PelletReasoner reasoner =   PelletReasonerFactory.getInstance().createNonBufferingReasoner(ontology);
manager.addOntologyChangeListener(reasoner);
reasoner.precomputeInferences();
/* Inference check performed, we should have resolved Object Properties: */
/* dmp ~> hasSecurityClassificationBreach ~> Computer1 */
/* Computer1 ~> isOwnedBy ~> dmp */
/* Computer1 ~> hasSecurityClassificationBreach ~> dmp */
```

At this point, we do not investigate the results. Let's proceed by adding new individuals and relationships. For the purposes of illustration, what we do here is replicating the example process of the previous section, that is, programmatically adding a new User with a relationship to a new Computer that will trigger the SWRL rule defined within the ontology.

```
/* Add a new user and computer, each with their classification */
OWLClass concept = factory.getOWLClass(IRI.create(ontolocation + "#User"));
OWLNamedIndividual user = factory.getOWLNamedIndividual(IRI.create(ontolocation + "#txa"));
manager.applyChange(new AddAxiom(ontology, factory.getOWLClassAssertionAxiom(concept, user)));
OWLNamedIndividual classification = factory.getOWLNamedIndividual(IRI.create(ontolocation +
      "#RESTRICTED"));
OWLObjectProperty role = factory.getOWLObjectProperty(IRI.create(ontolocation + "#hasClassification"));
manager.applyChange(new AddAxiom(ontology, factory.getOWLObjectPropertyAssertionAxiom(role, user,
      classification)));

concept = factory.getOWLClass(IRI.create(ontolocation + "#Computer"));
OWLNamedIndividual computer = factory.getOWLNamedIndividual(IRI.create(ontolocation +
      "#Computer2"));
manager.applyChange(new AddAxiom(ontology, factory.getOWLClassAssertionAxiom(concept, computer)));
classification = factory.getOWLNamedIndividual(IRI.create(ontolocation + "#SECRET"));
```

```
manager.applyChange(new AddAxiom(ontology, factory.getOWLObjectPropertyAssertionAxiom(role,
        computer, classification)));

/* Make the new user own the new computer to create a breach */
role = factory.getOWLObjectProperty(IRI.create(ontolocation + "#owns"));
manager.applyChange(new AddAxiom(ontology, factory.getOWLObjectPropertyAssertionAxiom(role, user,
        computer)));
```

Time now to extract inferences. We can do this by telling Pellet to give us what it learnt, using a filter to target only those inferences that we may be interested in. In this case, we know that we only need to find security breaches that are represented as Object Properties.

```
/* Examine inferences by attaching a generator to Object Properties */
List<InferredAxiomGenerator<? extends OWLAxiom>> generators;
OWLOntology inferredOntology = manager.createOntology();
generators = new ArrayList<InferredAxiomGenerator<? extends OWLAxiom>>();
generators.add(new InferredPropertyAssertionGenerator());
InferredOntologyGenerator generator = new InferredOntologyGenerator(reasoner, generators);
generator.fillOntology(manager, inferredOntology);
```

The newly inferred content in the ontology would normally be a small portion of the complete ontology. Using the API, we can iterate through this subset and search for anything that we are particularly interested in. Note that we have knowledge of the individuals we inserted, so we can further filter our result set, which will also contain inferences that were found previously and are still valid. Here is the check for the hasSecurityClassificationBreach Object Property for the newly inserted User.

```
/* Ask the reasoner for the breach property values for the new user */
OWLObjectProperty hasBreach = factory.getOWLObjectProperty(IRI.create(ontolocation +
        "#hasSecurityClassificationBreach"));
NodeSet<OWLNamedIndividual> breachValuesNodeSet = reasoner.getObjectPropertyValues(user,
        hasBreach);
Set<OWLNamedIndividual> values = breachValuesNodeSet.getFlattened();
System.out.println("Breach property values for User txa are: ");
for(OWLNamedIndividual individual : values) {
   System.out.println("    " + individual);
}
```

The printout from the above code will contain the OWL reference of the Computer individual that was added earlier (**Computer2**). In our implementation, the OWL definition file is located on the local drive and is therefore referenced as a file resource.

```
Breach property values for User txa are:
   <file:/cyber1.owl#Computer2>
```

The API also provides functions to produce the complete inference set as an OWL file, which can then serve as the source for conversion back to Cyber1 for an SV bridge.

6. Final Thoughts

We have outlined the steps necessary to successfully augment our existing reasoning capability using a Web Ontology standard and freely available tools. These included the

- translation of our existing ontology into OWL,
- the creation of SWRL rules that specify the criteria that we would like to be able to observe when they are met, and
- programmatically modifying the ontology and observing new inferences triggered by these rules.

Note that we did not provide any specific technical details on how the above can be integrated into Shapes Vector as we solely concentrated on the OWL side of things. It is envisaged that the supplementary process to enable reasoning with OWL would be trivially supported by an appropriate ontology bridge that converts Cyber1 to and from OWL using the guidelines from the ontology implementation section of this paper.

Whilst this study opens up reasoning opportunities that may not currently be available on our existing platforms, several questions could be asked. We did not, for example, test the performance of our solution. However, the size of our ontology is small, and because this system is designed to be employed at the higher levels in our agent architecture, the size of the input per time unit is also likely to be magnitudes smaller than what is generally observed at the lower levels. Therefore, we do not expect the number of inferences that will be made to put great strain on now commonly available hardware.

The SWRL rule we produced also highlighted some shortcomings that may be resolved with a different approach or future enhancements to the language. We had to refer to individuals rather than a Class concept when constructing the rule. It was also not possible to create a rule that could cover classifications in a generic fashion, meaning that separate rule would need to be created for **RESTRICTED** versus **SECRET**, **RESTRICTED** versus **CONFIDENTIAL**, and so on**.** It would be nice to be able to resolve such situations by the use of a variable in a single rule instead.

Finally, we noted earlier that we used an Object Property instead of Class to detect a breach. This was due to individuals not allowed to be created by rules. A Class would have been a preferred solution as it allows much richer and natural expressiveness than a property. For example, we could define generic hierarchies of vulnerabilities and exploits (as Classes) and SWRL rules then create an individual (Class instance) as consequence of the rule antecedents matching. Creating individuals thus provides separation between the domain ontology and the security ontology, freeing the user from manipulating Object Properties in the domain ontology in order to define security breaches. Fortunately, some of this expressiveness is still available with Object Properties. We can, for example, have hierarchies of Object Properties. Therefore, we could create one for each breach and make them all subclasses of a generic Breach property. This could then be used to filter for all inferred breaches through a single concept.

References

[1] UniSec Ontology Definition, Unified Security Discipline, C3ID, DSTO, http://c3idthewok.dsto.defence.gov.au:8080/display/USS/UniSec%20Ontology
[2] Matthew Horridge, "A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools", Edition 1.2, The University of Manchester, March 13, 2009
[3] Shapes Vector Development Software Development (SVSD) Project, "Ontology [Level 1] Description", Issue 1.4, October 26, 2007
[4] http://www.w3.org/2007/OWL/wiki/Implementations, sighted 5 October 2011
[5] Proposal for OWL-Based Incident Detection, Unified Security Discipline, C3ID, DSTO, http://c3idthewok.dsto.defence.gov.au:8080/display/USS/Proposal+for+OWL-Based+Incident+Detection
[6] OffSec Incident Dectection, Unified Security Discipline, C3ID, DSTO, http://c3idthewok.dsto.defence.gov.au:8080/display/USS/OffSec+Incident+Detection
[7] SWRL: A Semantic Web Rule Language Combining OWL and RuleML, Ian Horrocks et al., W3C, 2004, http://www.w3.org/Submission/SWRL/
[8] http://swrl.stanford.edu/ontologies/built-ins/3.3/swrlx.owl sighted 30 June 2011
[9] http://protege.cim3.net/cgi-bin/wiki.pl?SWRLExtensionsBuiltIns, sighted 30 June 2011
[10] http://answers.semanticweb.com/questions/2093/can-new-individuals-or-literals-be-inserted-using-rules sighted June 30 2011
[11] SPARQL 1.1. Query Language, W3C, http://www.w3.org/TR/2011/WD-sparql11-query-20110512/
[12] OffSec Data Fusion, Unified Security Discipline, C3ID, DSTO, http://c3idthewok.dsto.defence.gov.au:8080/display/USS/OffSec+Data+Fusion
[13] Pellet: OWL2 Reasoner for Java, Clark&Parsia, http://clarkparsia.com/pellet/

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

Testing the Shapes Vector derived UniSec Ontology in OWL using Protégé and Pellet

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Tamas Abraham and Dean Philp

5. CORPORATE AUTHOR

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TN-1052	AR-015-172	Technical Note	dECEMBER 2011

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
2011/1226380/1	N/A	N/A	11	13

DSTO Publications Repository

http://dspace.dsto.defence.gov.au/dspace

14. RELEASE AUTHORITY

Chief, Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS Yes

18. DSTO RESEARCH LIBRARY THESAURUS

Ontology, reasoning, programming

19. ABSTRACT

This document describes the details of implementing the UniSec ontology [1] in the Web Ontology Language (OWL) using the Protégé ontology development tool, and testing it with the Pellet reasoning engine to evaluate its usefulness.