Using SWRL and OWL to Capture Domain Knowledge for a Situation Awareness Application Applied to a Supply Logistics Scenario

Christopher J. Matheus[1], Kenneth Baclawski[2], Mieczyslaw M. Kokar [2], and Jerzy J. Letkowski[3]

[1] Versatile Information Systems, Inc.
Framingham, Massachusetts USA
cmatheus@vistology.com
http://www.vistology.com
[2] Northeastern University
Boston, Massachusetts USA
ken@baclawski.com mkokar@ece.neu.edu
[3] Western New England College
Springfield, MA, USA
jletkows@wnec.edu

Abstract. When developing situation awareness applications we begin by constructing an OWL ontology to capture a language of discourse for the domain of interest. Such an ontology, however, is never sufficient for fully representing the complex knowledge needed to identify what is happening in an evolving situation – this usually requires general implication afforded by a rule language such as SWRL. This paper describes the application of SWRL/OWL to the representation of knowledge intended for a supply logistics scenario. The rules are first presented in an abstract syntax based on n-ary predicates. We then describe a process to convert them into a representation that complies with the binary-only properties of SWRL. The application of the SWRL rules is demonstrated using our situation awareness application, SAWA, which can employ either Jess or BaseVISor as its inference engine. We conclude with a summary of the issues encountered in using SWRL along with the steps taken in resolving them.

1 Introduction

The problem of Situation Awareness involves the context-dependent analysis of the characterization of objects as they change over time in an evolving situation with the intent of establishing an understanding of "what is going on". Classic examples of tasks where situation awareness is of great importance include air traffic control, crisis management, financial market analysis and military battlespaces. For domains such as these, significant effort has gone into both understanding the problem and developing automated techniques and applications for establishing situation awareness[1]. A key part of this problem is identifying relations among the objects that are relevant to the situation and to the goals of the situation analyst. For any non-trivial situation the number of possible relations that might be considered is so vast that it is necessary to reduce the space of candidate relations by using additional knowledge about the situation's domain and about the specific objectives of the current situation. In an automated system designed to assist in establishing situation awareness this additional knowledge falls under the broad heading of "domain knowledge" and its use requires some form of knowledge representation (KR).

In our work on developing situation awareness systems we have been exploring the use of Semantic Web technologies for domain knowledge representation. We have found the OWL Web Ontology Language to be a very useful means for capturing the basic classes and properties relevant to a domain. These domain ontologies establish a language of discourse for eliciting more complex domain knowledge from subject matter experts (SME). Due to the nature of OWL, these more complex knowledge structures are either not easily represented in OWL or, in many cases, are not representable in OWL at all. The classic example of such a case is the relationship uncleOf(X,Y). This relation, and many others like it, requires the ability to constrain the value of a property (brotherOf) of one term (X) to be the value of a property (childOf) of the other term (Y); in other words, the siblingOf property applied to X (i.e., brotherOf(X,Z)) must produce a result Z that is also a value of the childOf property when applied to Y (i.e., childOf(Y,Z). This "joining" of relations is outside of the representation power of OWL. One way to represent knowledge requiring joins of this sort is through the use of the implication (→) and conjunction (AND) operators found in rule-based languages. The rule for the uncleOf relationship appears as follows:

```
brotherOf(X,Z) AND childOf(Y,Z) → uncleOf(X,Y)
```

We initially started developing the complex knowledge structures needed for our situation awareness applications using RuleML[2] as described in[4]. With the introduction of the Semantic Web Rule Language[3] (SWRL) we decided to investigate the potential for its use in our applications. SWRL was attractive because of its close connection with OWL DL and the fact that it, like OWL, has well-defined semantics. The two biggest drawbacks we saw at the time were its restriction to binary predicates (a characteristic inherited from OWL) and the lack of tools, in particular editors and consistency checkers. We confronted the lack of tools in part by developing our own graphical editor for SWRL called RuleVISor, but there is still an outstanding need for tools to check for consistency 1) within SWRL rules, 2) across SWRL rules and 3) between SWRL rules and the OWL ontologies upon which they are built. As for the issue of binary predicates we employ an approach by which n-ary predicates, such as the unconstrained predicates permitted by RuleML, can be systematically converted into binary predicates represented in SWRL; we describe this approach in Section 5 of this paper.

As we worked further on developing and using SWRL rules we encountered a number of additional issues that needed to be addressed before a practical implementation of our application could be realized. These issues include 1) the lack of negation as failure, 2) the need for procedural attachments and 3) the implementation of SWRL built-ins. Other concerns of particular importance to situation awareness – such as the representation of time, data pedigree and uncertainty – are not explicitly addressed in either SWRL or OWL; in fact, for the case of time (more specifically the changes of property values over time) the languages' monotonicity assumption technically precludes them or at least requires significant extra effort to circumvent the imposed constraints. We reported on some of these issues in our position paper and presentation[4] at the W3C Workshop on Rule Languages for Interoperability[5] and further elaborate on them in this paper.

The primary intent of this paper is to describe our experience of using SWRL and OWL to represent the domain knowledge for a supply logistics scenario and show how this knowledge was employed in our situation awareness application, SAWA.[6,7] We begin the paper by introducing the supply logistics "repairable assets" scenario and then describe the OWL ontology we developed to capture the scenario's key classes and properties. We then describe the domain knowledge rules for the scenario, starting with an abstract set of higher-order rules (i.e., rules that permit n-ary predicates) that are relatively easy to understand. These rules are then converted into an abstract representation in which the n-ary predicates have been converted to instances of classes representing the predicates and properties corresponding to the n-ary terms. These abstract rules are then converted into the less easy to read SWRL syntax. The SWRL rules are made operational by translating them into rules appropriate for interpretation by a forward-chaining inference engine – a process requiring additional operators such as `gensym`, `assert` and procedures to implement SWRL built-ins. The processing of these rules by SAWA is briefly summarized and a performance comparison is made between the use of two inference engines, BaseVISor (a Rete-based inference engine optimized for triples) and Jess (a Java implementation of the Rete-based CLIPS inference engine), either of which can be plugged into SAWA.

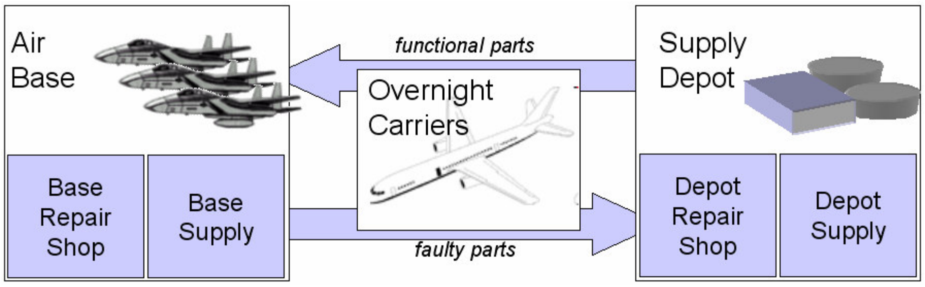
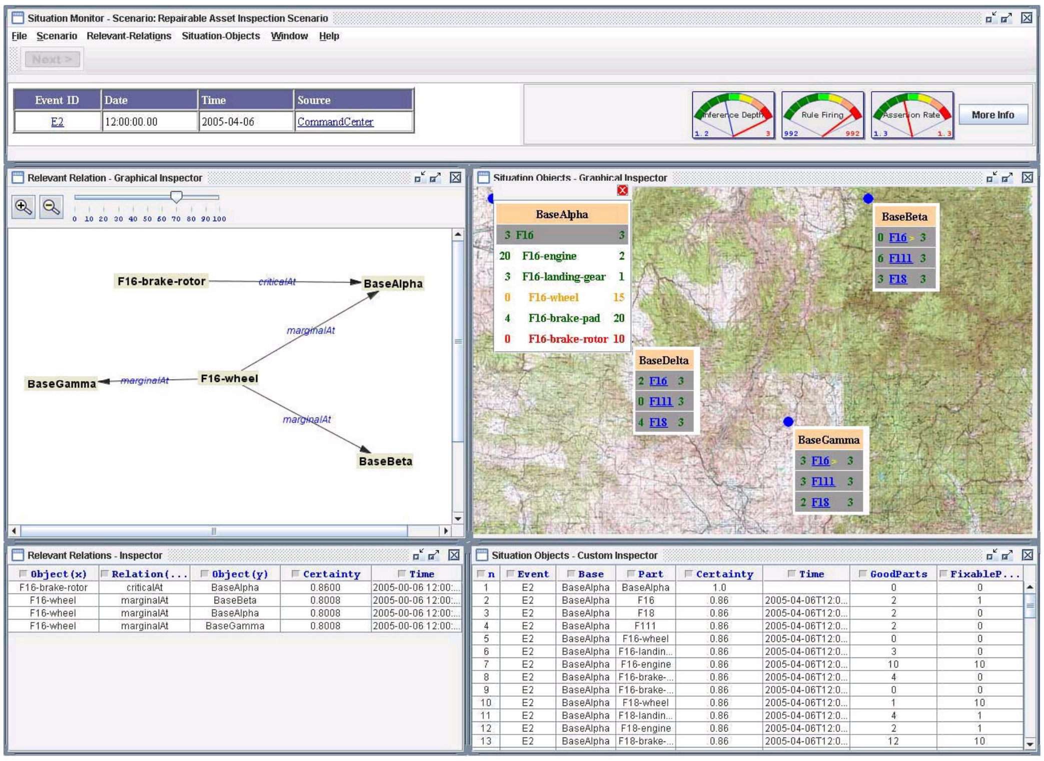


Figure 1. Repairable Assets Pipeline

2 Repairable Assets Domain

With assistance from SMEs at AFRL Wright Research Site we analyzed a supply logistics scenario involving the monitoring of "repairable assets". Repairable assets for the USAF represent aircraft parts that when found to be malfunctioning on an aircraft can be repaired for reuse either locally at the airbase's repair shop or at a remote repair depot. The diagram in Figure 1 shows a simplified version of the repairable assets pipeline used by the USAF. Each airbase has a supply of aircraft parts maintained at its local base supply along with the capability of repairing certain types of parts at its local repair shop. Some parts cannot be repaired locally and so they are shipped (usually by commercial overnight carriers) to remote supply depots that have more extensive repair capabilities. The supply depots repair whatever parts they can and place them into their depot supplies from which they are shipped out to airbases as needed. Keeping repairable parts at a sufficient level across a collection of airbases is a complicated process as it involves an understanding of the aircraft based at each facility, the repairable parts needed by each aircraft type, the repair capabilities of each base and the supply levels and repair capacity of all remote supply facilities.



Event ID	Date	Time	Source
E2	12:00:00.00	2005-04-06	CommandCenter

Object (x)	Relation(...	Object (y)	Certainty	Time
F16-brake-rotor	criticalAt	BaseAlpha	0.8600	2005-00-06 12:00:...
F16-wheel	marginalAt	BaseBeta	0.8008	2005-00-06 12:00:...
F16-wheel	marginalAt	BaseAlpha	0.8008	2005-00-06 12:00:...
F16-wheel	marginalAt	BaseGamma	0.8008	2005-00-06 12:00:...

n	Event	Base	Part	Certainty	Time	GoodParts	FixableP...
1	E2	BaseAlpha	BaseAlpha	1.0		0	0
2	E2	BaseAlpha	F16	0.86	2005-04-06T12:0...	2	1
3	E2	BaseAlpha	F18	0.86	2005-04-06T12:0...	2	0
4	E2	BaseAlpha	F111	0.86	2005-04-06T12:0...	2	0
5	E2	BaseAlpha	F16-wheel	0.86	2005-04-06T12:0...	0	0
6	E2	BaseAlpha	F16-landin...	0.86	2005-04-06T12:0...	3	0
7	E2	BaseAlpha	F16-engine	0.86	2005-04-06T12:0...	10	10
8	E2	BaseAlpha	F16-brake-...	0.86	2005-04-06T12:0...	4	0
9	E2	BaseAlpha	F16-brake-...	0.86	2005-04-06T12:0...	0	0
10	E2	BaseAlpha	F18-wheel	0.86	2005-04-06T12:0...	1	10
11	E2	BaseAlpha	F18-landin...	0.86	2005-04-06T12:0...	4	1
12	E2	BaseAlpha	F18-engine	0.86	2005-04-06T12:0...	2	1
13	E2	BaseAlpha	F18-brake-...	0.86	2005-04-06T12:0...	12	10

Figure 2. SAWA Interface for the Repairable Assets Domain

The specific objective of our application was to demonstrate the ability to effectively monitor the supply levels across a handful of airbases kept in supply by a set of remote supply depots. While it would certainly be possible to develop a one-time, stand-alone application to achieve this task we were interested in demonstrating the applicability of our general-purpose situation awareness application, SAWA, to this problem. SAWA uses OWL and SWRL to represent knowledge relevant to a domain of situation awareness problems and then employs a generic inference engine, BaseVISor or Jess, to reason about a specific evolving situation. SAWA has been described elsewhere[6,7] and so we will not go into the details of its workings in this paper. To provide a glimpse into what SAWA does, a screenshot of its interface adapted for the Repairable Assets domain is shown in Figure 2. The interface is comprised of five windows. The top-most window is the control pane in which resides the control menus, current event info and performance meters. In the middle right-hand window, a map depicts the physical locations of the airbases along with drillable summary sub-windows showing the status of the planes and parts present at each base. The middle left-hand window provides an interactive, graphical representation of the relations detected in the situation; these relations are also described in detail in the relations table that appears in the lower left-hand window. Detailed information about all of the objects and object attributes appear in the object table in the lower right-hand window.

3 Repairable Assets Ontology

As is our practice[4], we began the process of capturing the domain knowledge pertinent to the repairable assets problem by first developing an ontology in OWL. The primary objects in this ontology (shown in Figure 3) include airbases, airplanes, parts, facilities and remote supply depots. Because we are concerned with quantities of items such as parts and aircraft it was also necessary to create a QuantityOf class that permits the association of a numeric count with a specific plane or part type. In this way we can say that a particular facility has a QuantityOf instance relating a particular item with a specific number. It was also necessary to be able to associate each airbase with an ordered list of remote supply facilities available to provide additional parts; as shown, this was achieved using an rdf:List structure.

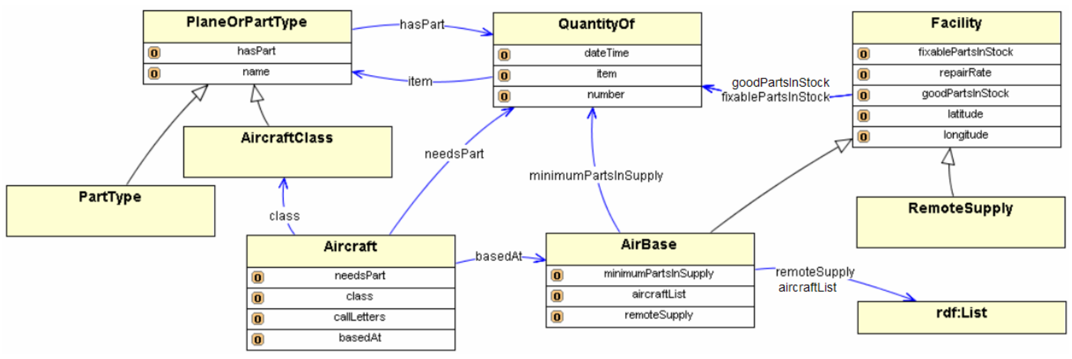


Figure 3. Repairable Assets Ontology

Simulated data was constructed for this scenario consisting of the inventory of aircraft and parts at four airbases and three remote supply bases taken at various times. This data was annotated using both the Repairable Assets ontology and the Event ontology shown in Figure 4. Each event contained facility-specific information such as the quantity of good aircraft of each type, the quantity of aircraft parts in stock, and the quantity of fixable parts in stock along with the current need for parts that needed to be replaced on aircraft undergoing repair. In addition to this “event” data, a file of annotations was created containing descriptions of the various aircraft types and the parts that make them up, while another annotation file was constructed to provide descriptions of the specific airbases, their aircraft and their remote supply facilities.

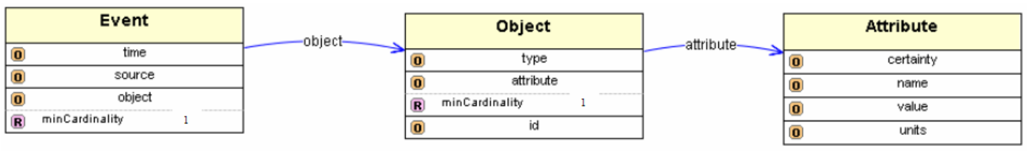


Figure 4. Event Ontology

4 Repairable Assets Domain Rules

The objective in our repairable assets scenario is to monitor the supply levels of various parts at a number of airbases and compare them to the current needs for those parts by specific aircraft. We developed an initial set of SWRL rules to achieve this using RuleVISor, a graphical rule editor we developed at Versatile Information Systems. These rules deal with local and remote supply levels, local demand levels and repair rates on a per-part and per-facility basis, identifying when a specific part-type at a specific airbase is "critical", "marginal" or "nominal". In all there were nine rules, some of them recursive, that were developed for this task. The logic captured by these rules is shown here using an abstract Prolog-like Horn-clause representation, in which variables are prefaced with question marks:

```
criticalPartAtFacility(?Part, ?Facility, ?Time) :-
  localNeed(?Part, ?Facility, ?Time, ?Need)
  localSupply(?Part, ?Facility, ?Time, ?Supply)
  ?Need <= ?Supply.

 marginalPartAtFacility (?Part,?Facility,?Time) :-
  localSupply(?Part, ?Facility, ?Time, ?Supply)
  localRepairable(?Part, ?Facility, ?Time, ?Repairable)
  remoteAvailable(?Part, ?Facility, ?Time, ?RemoteSupply)
  surplusRequired(?Part, ?Facility, ?SurplusRequired)
  ?Supply + ?Repairable + ?RemoteSupply < ?SurplusRequired).
```

In plain English these rules state that a part at a facility is determined to be "critical" if the current demand at the facility exceeds the current local supply; it is classified as "marginal" if the total resuppliable rate for the part at the facility is below a required-surplus threshold; and it is deemed "nominal" otherwise (note, there are no rules for this state as it is the normal state of all parts that are neither marginal nor critical). The notion of "resuppliable rate" used in the marginalPartAtFacility rule represents the total number of parts of a specific type that a facility could have on hand by the next day if its current local supply level of that part, its current local "repair capacity" for that part and the current supply levels of that part at remote depots are all added together. If this total falls below the required-surplus level (a static number established by an SME) the part at that facility is given a status of "marginal'. It is common practice in the USAF to ship parts as needed between facilities by overnight delivery, which leads to the natural choice of a day as the basis for determining part resuppliability. The "repair capacity" at an airbase is a function of the number of parts waiting to be repaired locally, the repair capacity of the local repair shop and the status of any sub parts required for the repairs (this value is calculated by additional supporting rules not shown above).

5 Converting from N-Ary to Binary Predicates

As is evident in the abstract rules for criticalPartAtFacility and marginalPartAtFacility, we used predicates with more than two terms. We did this because it was the natural way to represent the critical concepts, all of which simultaneously involve a Part, a Facility and a Time; unfortunately such n-ary predicates are not permitted in SWRL. As a result, we needed to convert these rules into ones that contained only binary and unary predicates. The presentation of two design patterns usable for this purpose have been described by Noy and Rector[8]; our approach is in line with the second of these patterns. This conversion was done manually for this small set of rules but the process we employed is systematic enough to automate; for a set of rules defined in RuleML a single XSLT script would suffice. The approach involves converting the n-ary predicates into instances of unique classes (one for each predicate) that are then given properties corresponding to the each of their respective terms. The results of this process can be seen in the following rule corresponding to the marginalPartAtFacility rule described above.

```
if
    ;; find the Local Surplus/Deficit (from another rule)
    rdf:type(?SMNStatement, #SupplyMinusNeed)
    #smnPart(?SMNStatement, ?Part)
    #smnFacility(?SMNStatement, ?Facility)
    #smnTime(?SMNStatement, ?Time)
    #smnNumber(?SMNStatement, ?LocalSurplusOrDeficit)

    ;; find the number Locally Repairable (from another rule)
    rdf:type(?PLRStatement, #PartsLocallyRepairable)
    #localPart(?PLRStatement, ?Part)
    #localFacility(?PLRStatement, ?Facility)
    #localNumber(?PLRStatement, ?NumberRepairable)

    ;; find number Available Remotely (from another rule)
    rpa:remoteSupply(?Facility, ?RemoteSupplyList)
    rdf:type(?PRAStatement, #PartsAvailableAtRemoteFacility)
    #remotePart(?PRAStatement, ?Part)
    #facilityList(?PRAStatement, ?RemoteSupplyList)
    #remoteNumber(?PRAStatement,?NumberAvailableRemotely)
    #remoteTime(?PRAStatement, ?Time)

    ;; look up Minimum Threshold
    rpa:minimumPartsInSupply(?Facility,
                             #MinimumThresholdStatement)
    rpa:item(?MinimumThresholdStatement, ?Part)
    rpa:number(?MinimumThresholdStatement, ?MinimumThreshold)

    ;; add SurplusOrDeficit, Repairable, & RemotelyAvailable
    swrlb:add(?TotalAvailable, ?LocalSurplusOrDeficit,
        ?NumberRepairable, ?NumberAvailableRemotely)

    ;; test if the Threshold is greater than the Total
    swrlb:greaterThan(?MinimumThreshold, ?TotalAvailable)

then
    rdf:type(?MPFStatement, #MarginalPartAtFacility)
    #marginalPart(?MPFStatement, ?Part)
    #marginalFacility(?MPFStatement, ?Facility)
    #marginalTime(?MPFStatement, ?Time)
    #marginalNumber(?MPFStatement, ?TotalAvailable)
```

In the conclusion of the rule (i.e., the five lines of code after the "then") it can be seen that the marginalPartAtFacility(?Part,?Facility,?Time) predicate has been converted into an instance of a locally defined class MarginalPartAtFacility represented by the variable ?MPFStatement. To this instance three properties (marginalPart, marginalFacility and marginalTime) have been attributed with values corresponding to the variables ?Part, ?Facility and ?Time. (A fourth property "marginalNumber" is used to encode the degree to which the required surplus level has been unmet.) In the body of the rule there are three places where the predicates localSupply, localRepairable and remoteSupply from the abstract rule have been converted into statements referring to instances of local classes with properties corresponding to each of their original four terms. For example, the localSupply(?Part,?Facility,?Time,?Supply) predicate from the abstract rule is converted into a reference to an instance of the class SMNStatement (SMN stands for Supply Minus Need) which has the four properties - localPart, localFacility, localTime and localNumber - associated with the four terms corresponding to Part, Facility, Time and Supply, respectively. When this rule is processed, the inference engine will look for the occurrence of these class instances and their corresponding properties in working memory; it will only find them if other rules (not shown) have previously fired and as a result asserted these instances and properties.

Note that the classes used to stand in place of the n-ary predicates are all defined local to the rule set (as indicated by the preceding hash mark #) and are not a part of the main ontology described in Section 3. These classes do not in fact have to be explicitly defined in the rule set because the mere use of one as the object of the rdf:type property requires (by the axioms of RDF/OWL) that there be a local class identified by that reference; any OWL-compliant reasoner will infer its existence. The same holds true for the properties that stand in place of the predicate terms. The only requirements of these local classes are that they be uniquely named and that whenever one is used in the body of a rule that there be at least one rule that asserts an instance of the same class in its head, otherwise the first rule will never fire.

6 SWRL Rules and Their Execution

The SWRL code for the marginalPartAtFacility rule, developed with the help of RuleVISor, appears in Appendix A. The primary reason for including the SWRL code in this paper is to demonstrate how a relatively simple rule expressible in just six lines of high-level code mushrooms into a much more complex listing of over one hundred lines of code that is extremely difficult to interpret and even more difficult to debug. The nine rules making up the SWRL rule set for the Repairable Assets scenario amounted to nearly 1200 lines of code and demanded countless hours of debugging. One can argue that SWRL was never intended to be a language for the manual development of rules and that what is needed are more powerful editors that permit rules to be represented more abstractly and then compiled into SWRL for execution. Neither RuleVISor nor the SWRL editor provided with the Protégé OWL plug-in[9] provide this level of functionality – both simply permit the direct editing of SWRL code with its inherent constraint to binary predicates. The requirement to work at such a painfully low level of representation is a major hurdle for anyone wishing to use SWRL for even moderately complex tasks.

Assuming one has a set of SWRL rules, such as the repairable assets rules described above, there remains the question of how to execute them. There are no inference engines known to the authors that have full native support for SWRL rules. The Institut für Informatik, Freie Universität Berlin[10] has developed a prototype engine for SWRL but it does not (as of this writing) permit full OWL reasoning (only inheritance reasoning is supported) nor does it implement the SWRL built-ins. We have implemented a Jess-based reasoner that includes a reasonably complete set of axioms for RDF/OWL and supports a large subset of SWRL built-ins; this reasoner is at the heart of our consistency checking service ConsVISor[11]. We have also recently implemented a high-performance Java-based inference engine that incorporates a subset of the axioms of RDF/OWL sufficient to support the reasoning required for our repairable assets problem domain as well as support for the SWRL built-ins used by these rules. To execute SWRL rules in either the Jess or BaseVISor engines it is first necessary to translate them into the corresponding rule languages. In doing so, engine specific characteristics need to be accounted for that lie outside the scope of SWRL. Both Jess and BaseVISor are Rete-based, forward chaining inference engines that work by continuously evaluating the contents of working memory and firing rules when their antecedents are satisfied. The firing of a rule can result in the "assertion" of new facts into working memory but this requires an explicit call to the "assert" operator. There is nothing in SWRL that corresponds to the assert operator – the atoms in the head of a SWRL rule are simply inferred to be true whenever all of the atoms in the body are true but there is no notion of "working memory" into which facts must be asserted. In the translation from SWRL to Jess or BaseVISor it is necessary to surround all atoms in the heads of rules with the assert() operator; this is done automatically by XSLT translation scripts.

Because of the use of instances of classes to represent n-ary predicates it is necessary to be able to "generate" these instances as needed. These instances in need of generation exist as variables in the head of rules that have no corresponding occurrence in the body of the rule. When such a variable is detected the translation scripts add a gensym() operator to the head to generate a unique symbol to represent the instance. At first this technique seems to be at odds with the "safety" condition in which all variables in the head of a SWRL rule must be present in the body, but as suggested in the SWRL specification, it would be possible to add a someValuesFrom restriction on this variable in the body; we don't actually do this because it would have no bearing at all on the firing of the rule.

There is also an issue with implementing the SWRL built-ins. The definitions of the built-ins in the SWRL specification are given as relations with no explicit input/output designations assigned to their arguments. In our rules we have always found that we need to use the built-in capabilities as if they were functions in which you specify the values for all but one of the arguments, which becomes the output variable. What this lack of input/output designation in SWRL built-ins means from an implementation perspective is that the code for the built-ins must determine which of the arguments is supposed to be the output variable and then select the appropriate functional method to apply to the other arguments. For example, consider swrlb:add which can be applied to an arbitrary number of two or more arguments with the first argument being the sum of the remaining arguments. If you use the atom (swrlb:add ?X 1 2) the processor of this statement must detect that the first argument is unbound (i.e. a variable) and thus it becomes the output argument. Knowing that the value of the first argument is to be calculated the processor must apply its *summation* method to the remaining arguments. If, on the other hand, you pass (swrlb:add 10 ?X 5) to the processor, it needs to figure out that it should *subtract* 5 from 10 to calculate the value of the variable ?X. In this case we have used swrlb:add to do *subtraction* even though there is also a swrlb:subtract built-in. Now consider the case where more than one argument is unbound: (swrlb:add ?X 100 ?Y). What should the processor do in this case? According to the semantics of SWRL this is perfectly legal even though it would result in an infinite set. In our system we treat this case as an error, which it would be for the kinds of practical applications we are interested in. Alternatively, if one really needed the set of all relations consistent with the bound terms, procedures could be implemented that return some notational form from which the members of the set could be derived. In our cases, however, this approach would needlessly complicate the rules and make it cumbersome to deal with the most common case where what is really desired is a function.

A final issue encountered with the practical application of SWRL was the need for some form of negation as failure. Since there are so many parts on an aircraft it was highly desirable to require airbases to only report when there was a need for the repair of a specific part rather than report the status of all parts on the aircraft. The quite natural assumption in this case is that a part is working properly unless informed otherwise. This becomes an issue because we need to be able to count the number of pairs needing repair at an airbase which requires looking up the number of parts needed for each aircraft in the airbase's rdf:List of stationed aircraft. As the rules walk through this aircraft list they can only fire if there is something in working memory that triggers them. That is unless the not() operator is used in which case the absence of a particular fact can lead to the firing of a rule, which is exactly what we want. In our rules that count the number of a specific part needing repair at an airbase there is one rule that checks for the occurrence of a fact stating that an aircraft at that base needs that part and another rule looking for the absence of such a fact. In our scenario this is a perfectly reasonable thing to do since airbases are required to report on all parts needing repair. We can thus safely assume that our world is closed within the scope of the status reports sent out by the airbases. Both BaseVISor and Jess provide a not() operator that implements negation as failure (NAF). Although neither of them provide a scoped NAF operator (SNAF) it would be easy to define the specific scope in this case using the URI specifying the Event that transmitted a base's status report data for a specific Time. Our rules actually implement a notion of Event scoping as most of them include an explicit reference to an Event for a specific Time (see the beginning of the rule listed in Appendix A).

7 BaseVISor versus Jess

We have used Jess as an inference engine for a number of RDF/OWL applications over the last couple years. On several occasions we developed code to extend its capabilities, most markedly in the area of SWRL built-ins and performance monitoring. When working with the internals of the Jess source code it becomes apparent that a lot of legacy code exists that is there to support the LISP-like list structures that CLIPS and OPS-5 supported. Since our RDF/OWL applications deal only with triples there is no need for all of the complexity enabled by Jess' data structures, which carry with them significant overhead particularly when doing lookups within the Rete network. When it came time to augment the Rete network with uncertainty processing capabilities (an important requirement for many situation awareness tasks) we took the opportunity to develop a triples-based Rete network from scratch. The result is BaseVISor, which has now replaced Jess as the core of SAWA. Space limitations prevent us from going into more details of the internals of BaseVISor or its support for SWRL; however, we do want to highlight the performance improvement afforded by BaseVISor over Jess as depicted in Figure 5. This graph compares the performance of BaseVISor versus Jess as the number of ground facts increases. Except for a very small portion at the lower left of the graph where BaseVISor is slightly slower (due to some index optimization that does not payoff on small data sets of less than 500 facts), BaseVISor outperforms Jess and shows a near linear rate of change compared to Jess' polynomial increase.

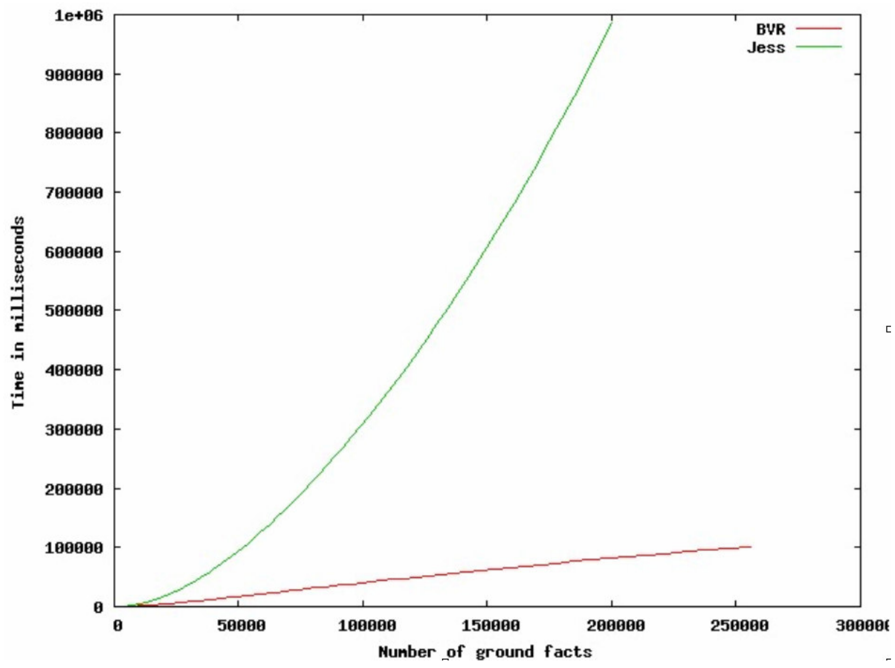


Figure 5. BaseVISor vs. Jess performance.

8 Conclusion

In our recent efforts to develop a situation awareness application for a supply logistics scenario we explored the utility of using SWRL and OWL to represent pertinent domain knowledge and apply it to simulated data using forward chaining inference engines. We encountered several challenges in applying SWRL to this problem including foremost the language's limitation to binary predicates and the lack of tools for editing and checking SWRL rules. We partially resolved the latter problem by developing RuleVISor, a graphical SWRL editor that permits the construction of rules using elements from OWL ontologies. Even with RuleVISor the restriction to binary predicates forced us to operate at a very low implementation level compared with the n-ary predicates that were more natural for representing the important domain concepts. Our approach to this problem was to develop abstract higher-arity rules by hand and then systematically convert the n-ary predicates into classes representing the predicates and collections of properties to associate values of the higher-arity terms with the predicate's class. While this was a manual process it is such that a simple translation script could perform the process automatically. The final steps of our rule development effort involved converting the abstract rules represented with binary-only predicates into SWRL syntax followed by the application of an XSLT script to translate the SWRL into either Jess or BaseVISor rules for their execution within the appropriate inference engine. We used Jess as our engine up until the completion of our own Rete-based inference engine, BaseVISor, which we have optimized for the processing of triples and incorporated support for uncertainty reasoning. Initial comparisons between the two engines demonstrated BaseVISor's near linear performance in the number of ground facts compared with Jess' polynomial performance.

Appendix A: SWRL Code for "Marginal Part at Facility" Rule

```
<ruleml:imp>
    <ruleml:_rlab
        ruleml:href="#Marginal part at facility"/>
    <ruleml:_body>
      <swrlx:classAtom>
        <owlx:Class owlx:name="&evt;Event"/>
        <ruleml:var>?Event</ruleml:var>
      </swrlx:classAtom>
      <swrlx:datavaluedPropertyAtom
          swrlx:property="&evt;time">
        <ruleml:var>?Event</ruleml:var>
        <ruleml:var>?Time</ruleml:var>
      </swrlx:datavaluedPropertyAtom>
      <swrlx:classAtom>
        <owlx:Class owlx:name="#SupplyMinusNeed" />
        <ruleml:var>?SMNStatement</ruleml:var>
      </swrlx:classAtom>
      <swrlx:individualPropertyAtom
          swrlx:property="#smnPart">
          <ruleml:var>?SMNStatement</ruleml:var>
          <ruleml:var>?Part</ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom
            swrlx:property="#smnFacility">
          <ruleml:var>?SMNStatement</ruleml:var>
          <ruleml:var>?Facility</ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:datavaluedPropertyAtom
            swrlx:property="#smnTime">
          <ruleml:var>?SMNStatement</ruleml:var>
          <ruleml:var>?Time</ruleml:var>
      </swrlx:datavaluedPropertyAtom>
      <swrlx:datavaluedPropertyAtom
           swrlx:property="#smnNumber">
          <ruleml:var>?SMNStatement</ruleml:var>
          <ruleml:var>?LocalSurplusOrDeficit</ruleml:var>
      </swrlx:datavaluedPropertyAtom>
      <swrlx:classAtom>
        <owlx:Class owlx:name="#PartsLocallyRepairable"/>
        <ruleml:var>?PLRStatement</ruleml:var>
      </swrlx:classAtom>
      <swrlx:individualPropertyAtom
            swrlx:property="#localPart">
          <ruleml:var>?PLRStatement</ruleml:var>
          <ruleml:var>?Part</ruleml:var>
```

```
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="#localNumber">
    <ruleml:var>?PLRStatement</ruleml:var>
    <ruleml:var>?NumberRepairable</ruleml:var>
</swrlx:datavaluedPropertyAtom>
    <ruleml:var>?PLRStatement</ruleml:var>
    <ruleml:var>?Facility</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
     swrlx:property="&rpa;remoteSupply">
    <ruleml:var>?Facility</ruleml:var>
    <ruleml:var>?RemoteSupplyList</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:classAtom>
   <owlx:Class
     owlx:name="#PartsAvailableAtRemoteFacility" />
  <ruleml:var>?PRAStatement</ruleml:var>
</swrlx:classAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#remotePart">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?Part</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#facilityList">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?RemoteSupplyList</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="#remoteNumber">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?NumberAvailableRemotely
    </ruleml:var>
</swrlx:datavaluedPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="#remoteTime">
    <ruleml:var>?PRAStatement</ruleml:var>
    <ruleml:var>?Time</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="&rpa;minimumPartsInSupply">
    <ruleml:var>?Facility</ruleml:var>
    <ruleml:var>?MinimumThresholdStatement
    </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom
      swrlx:property="&rpa;item">
    <ruleml:var>?MinimumThresholdStatement
    </ruleml:var>
    <ruleml:var>?Part</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datavaluedPropertyAtom
      swrlx:property="&rpa;number">
    <ruleml:var>?MinimumThresholdStatement
    </ruleml:var>
    <ruleml:var>?MinimumThreshold</ruleml:var>
</swrlx:datavaluedPropertyAtom>
```

```
    <swrlx:builtinAtom swrlx:builtin="&swrlb;add">
           <ruleml:var>?TotalAvailable</ruleml:var>
           <ruleml:var>?LocalSurplusOrDeficit
           </ruleml:var>
           <ruleml:var>?NumberRepairable</ruleml:var>
           <ruleml:var>?NumberAvailableRemotely
           </ruleml:var>
    </swrlx:builtinAtom>
    <swrlx:builtinAtom
              swrlx:builtin="&swrlb;greaterThan">
            <ruleml:var>?MinimumThreshold</ruleml:var>
            <ruleml:var>?TotalAvailable</ruleml:var>
    </swrlx:builtinAtom>
  </ruleml:_body>
  <ruleml:_head>
    <swrlx:classAtom>
    <owlx:Class owlx:name="#MarginalPartAtFacility"/>
        <ruleml:var>?MPFStatement</ruleml:var>
    </swrlx:classAtom>
    <swrlx:individualPropertyAtom
          swrlx:property="#marginalPart">
        <ruleml:var>?MPFStatement</ruleml:var>
        <ruleml:var>?Part</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom
          swrlx:property="#marginalFacility">
        <ruleml:var>?MPFStatement</ruleml:var>
        <ruleml:var>?Facility</ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:datavaluedPropertyAtom
          swrlx:property="#marginalTime">
        <ruleml:var>?MPFStatement</ruleml:var>
        <ruleml:var>?Time</ruleml:var>
    </swrlx:datavaluedPropertyAtom>
    <swrlx:datavaluedPropertyAtom
          swrlx:property="#marginalNumber">
        <ruleml:var>?MPFStatement</ruleml:var>
        <ruleml:var>?TotalAvailable</ruleml:var>
    </swrlx:datavaluedPropertyAtom>
  </ruleml:_head>
</ruleml:imp>
```

References

[1] M. Endsley and D. Garland, *Situation Awareness, Analysis and Measurement,* Lawrence Erlbaum Associates, Publishers, Mahway, New Jersey, 2000.

[2] Rule Markup Language Initiative, http://www.ruleml.org/

[3] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004. http://www.daml.org/rules/proposal/

[4] C. Matheus, Using Ontology-based Rules for Situation Awareness and Information Fusion. Position Paper presented at the W3C Workshop on Rule Languages for Interoperability, April 2005. http://www.w3.org/2004/12/rules-ws/program2

[5] W3C Workshop on Rule Languages for Interoperability. http://www.w3.org/2004/12/rules-ws/

[6] C. Matheus, M. Kokar, K. Baclawski, J. Letkowski, C. Call, M. Hinman, J. Salerno and D. Boulware, SAWA: An Assistant for Higher-Level Fusion and Situation Awareness. In Proceedings of SPIE Conference on Multisensor, Multisource Information Fusion, Orlando, FL., March 2005.

[7] C. Matheus, M. Kokar, K. Baclawski, J. Letkowski, C. Call, M. Hinman, J. Salerno and D. Boulware, Lessons Learned From Developing SAWA: A Situation Awareness Assistant, FUSION'05, Philadelphia, PA, July, 2005.

[8] N. Noy and A. Rector, Defining N-ary Relations on the Semantic Web: Use With Individuals, W3C Working Draft 21, July 2004.

[9] SWRL Editor for Protégé with the OWL plugin. http://protege.stanford.edu/plugins/owl/swrl/

[10] Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, An Engine for SWRL rules in RDF graphs. http://www.inf.fu-berlin.de/inst/ag-nbi/research/swrlengine/

[11] ConsVISor Consistency Checking Service, http://www.vistology.com/consvisor/