DAMLJessKB: A Tool for Reasoning with the Semantic Web

Joseph B. Kopena and William C. Regli*

Geometric and Intelligent Computing Laboratory
Department of Computer Science
College of Engineering, Drexel University
3141 Chestnut Street
Philadelphia, PA, 19104
regli@drexel.edu
http://gicl.mcs.drexel.edu/

Abstract. We describe DAMLJessKB, a tool for reasoning with the DARPA Agent Markup Language (DAML) and performing inference on the Semantic Web. DAMLJessKB maps DAML's semantics into facts and rules for use in a production system, such as the Java Expert System Shell (Jess). This article presents our underlying methodology and provides a detailed example of how DAML-JessKB can be used to make decisions about DAML-encoded engineering design knowledge. We believe that tools like DAMLJessKB are needed to help realize the full potential of the Semantic Web and DAML.

1 Introduction

DAMLJessKB is a tool for reasoning with the Semantic Web. It is a description logic reasoner for performing inference with the DARPA Agent Markup Language (DAML). This is accomplished by using a production system to carry out the automated inferences entailed by the semantics of the DAML, Resource Description Framework Schema (RDF-S), and XML Schema: Datatypes (XSD) specifications. We show how DAMLJessKB can be used to implement the reasoning necessary to build Semantic Web applications and provide an example from the domain of engineering design.

The Semantic Web is a vision of the next generation World Wide Web, where all content is both machine and human interpretable. Currently, the Web is largely a presentation medium intended for human users [1,2]. While much of the existing work has focused on representation and semantics, DAMLJessKB addresses how to use these representations for making inferences. The representations and semantics for our current DAMLJessKB are based on the RDF W3C REC-rdf-syntax-19990222 proposed standard [3]. To define information structure, we use the proposed RDF-S standard CR-rdf-schema-20000327 [4] to create the basic relationships between classes of objects and properties. In addition, the proposed XSD standard REC-xmlschema-2-20010502 [5] is used to manipulate and reason on literal data (i.e., integer and floating point numbers, etc.) and define new classes of literal values. Lastly, DAML builds a description

* Also with the Department of Mechanical Engineering and Mechanics.

D. Fensel et al. (Eds.): ISWC 2003, LNCS 2870, pp. 628–643, 2003.
© Springer-Verlag Berlin Heidelberg 2003

logic language on top of RDF-S and XSD where statements are expressed using RDF. DAMLJessKB currently uses the March 2001 DAML+OIL specification [6]. Using this language, the structure of complex information can be described through relationships between classes and properties of objects while maintaining tractable reasoning.

Creating Semantic Web applications requires applying the semantics of these languages to make decisions based on the entailments of their formal definitions. DAMLJessKB utilizes the Java Expert System Shell (Jess) [7] developed at Sandia National Laboratories to carry out this reasoning. Jess and DAMLJessKB are Java-based and well suited for incorporation into applications, applets, and servlets. Presently, DAMLJessKB is being employed in a number of research projects. Users and applications include: Penn State University's Industrial Engineering Department (product/process ontologies), UMBC's Agents group (information retrieval) [8], Coca Cola (enterprise integration), SRI, and Lockheed Martin's Advanced Technology Laboratories.

This article describes the development and use of DAMLJessKB in Semantic Web applications. The following section discusses and motivates some of the possible inferences and the role of such a reasoning capability in Semantic Web software. Then, we briefly describe related work and give an description of the DAMLJessKB tool. Finally, we present some prototype applications of this tool and some conclusions.

2 Motivation

DAML is a formal knowledge representation language, enabling the ability to perform automated inference with DAML-encoded knowledge. The formal semantics for DAML [9,10] can either be used directly, such as incorporating the axioms into a general theorem prover, or indirectly, such as in DAMLJessKB. With these formal semantics there is a precise correct interpretation of the standards, so we can characterize nonstandard assumptions we make, prove the correctness of our reasoning, etc.

One class of automated inference this enables is the ability to perform basic consistency checking on input documents. This relates to many people making use of DAML as a schema-type language. This is useful, as XML DTDs, Schema, etc. are focused on the syntactic structure of the document. DAML allows you to impose a structure on information modeled in RDF. This frees you from having to worry about the particulars of how your document is transcribed, and lets you worry about the information present.

However, such automated inference can be used in more general, more powerful ways. For example, it can be used for knowledge translation as well as application-specific tasks. Translation comes from determining relationships between classes, properties, etc. Application tasks come from being able to perform reasoning on input information to do such things as determine relationships between classes and properties. This could also be used in translation, if there was an underlying base ontology.

Application Scenario: Engineering Design. Consider the problem of managing repositories of engineering designs, such as shown in Figure 1. Every design includes CAD data (i.e., the mechanical assembly model, joints, features, tolerances, etc), design intent and rationale, and documents describing the intended function and behavior of the artifact. Large engineering enterprises may have huge knowledge-bases of current and past

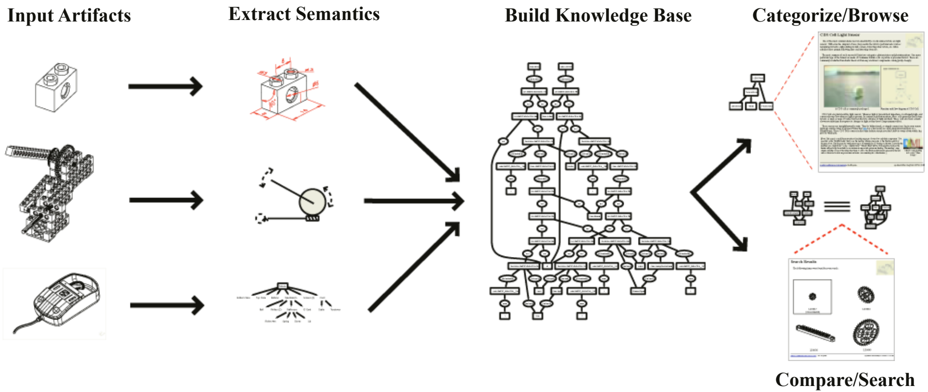


Fig. 1. Overview: Using DAMLJessKB to reason with engineering data

product information that engineers must interrogate throughout the product lifecycle. For example:

- Searching engineering repositories [11] to find models for use in variational design. Variational design is the task of adapting old designs to solve new requirements. This would require being able to make inferences about device behavior and function.
- Analyzing design and manufacturing data across different domains. This would require the specification of and inference with shared ontologies that capture the engineering or manufacturing semantics of a class of artifacts.
- Maintaining existing products, which requires being able to make inferences about stored design knowledge and design rationale.

It is our belief that DAML and similar languages could be the basis of new knowledge interchange standards that go beyond the current STEP standard for product data exchange (ISO 10303) [12]. Answering questions like those above requires more than the shared syntax of STEP. DAML-based representations that capture designs, parts and assemblies, tolerances, features, etc could form the basis of new representations that can be used in conjunction with automatic tools for inference and knowledge discovery. We show a detailed example of this sort of application in Section 6.

3 Related Work

In order to correctly input information modeled in DAML, applications need to implement the semantics of the language. This also enables the application to make use of the inferences entailed by those semantics in its own reasoning process. In this section we review work in providing reasoning for DAML and other Semantic Web languages.

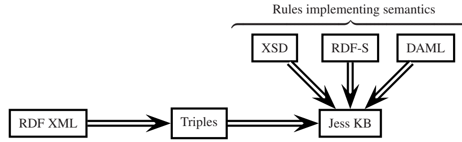
Description Logic Reasoners. DAML has been designed as a description logic (DL) language with a syntax suitable for being embedded in Web documents. Other efforts at providing inference tools for the Semantic Web have built translators to existing DL reasoners, such as FaCT [13]. Compared to such efforts, DAMLJessKB attempts to

support a broader set of relevant languages, including XSD, RDF-S, and DAML+OIL. DAMLJessKB's Java-based implementation also allows for easier incorporation into Java-based applications than many current DL systems. Notably, this provides a Semantic Web inference tool for use in applets, servlets, and Java-enabled embedded devices such as cell phones and PDAs. In addition, as opposed to most current DL implementations, DAMLJessKB does not draw a clear boundary between instances and terminologies in the knowledge base. More discussion on DAML and DAMLJessKB's support for reasoning on classes of classes is presented in Section 4.

A prominent distinction between DL reasoners and DAMLJessKB is that nearly all of the former make an open world assumption while the latter's use of an underlying production system entails a closed world assumption. We return to this issue in Section 4. DAMLJessKB's implementation approach does cost in terms of computational complexity as compared to results for basic description logics. However, reasoners supporting all of the DAML elements, such as `equivalentTo` and `TransitiveProperty`, will also suffer as compared to these results. In our experience, the expressiveness of the underlying production system has been sufficient for implementing the semantics of these languages in a closed world interpretation. We believe this to be demonstrable through the intersection of the subsets of first order logic corresponding to classes of description logics and production systems but present no analysis in this article.

Non-DL Reasoners. A variety of Semantic Web projects have taken an approach similar to DAMLJessKB of translating DAML semantics and content into forms suitable for use in non-description logic reasoners such as general first order theorem provers and production systems. A reasonably complete list of such efforts is presented at `http://www.daml.org/reasoning/`. Compared to DAMLJessKB, most of the efforts based on production systems have not attempted to implement as large a portion of the DAML semantics. Typically, they incorporate little terminological reasoning and support for the more expressive DAML elements. Few support XML Schema Datatypes. General first order theorem proving approaches often incorporate most of the more expressive DAML statements but not support for datatypes as many theorem provers are ill suited for reasoning on literal values. Such packages are subject to complexity results for the theorem prover, as well as the difficult problem often encountered in practice of tuning the system to control the search/search space and produce acceptable runtimes. Additonally, nearly all theorem provers operate under an open world assumption, an issue to which we will return in Section 4.

Programmatic Tools. Many initial and current Semantic Web-related applications and demonstrations simply operate on the XML structure of input data, or construct APIs based on an object-oriented view of RDF data. In these cases, any ontology developed for the application is misused as a schema for the XML data, or as a specification similar to class definitions in object-oriented programming languages, ignoring most of the semantics. Such approaches are, of course, limited in their ability to operate with other applications and data sources as they will only be able to process a small set of valid input documents. Constructing APIs for DAML will either suffer the same problem through ignoring most of the semantics or will effectively implement a description logic reasoner. In either case, when the semantics are ignored, the application suffers through limitations



(a) Overview of DAMLJessKB process

```
<rdfs:Class rdf:about="#Artifact" />

<rdfs:Class rdf:about="#Part">
  <rdfs:subClassOf rdf:resource="#Artifact"/>
</rdfs:Class>

<eng:Part rdf:about="#Cog17" />
```

(b) RDF snippet in XML form

```
⟨ ⟨rdf:type eng:Artifact rdfs:Class⟩
⟨rdf:type eng:Part rdfs:Class⟩
⟨rdfs:subClassOf eng:Part eng:Artifact⟩
⟨rdf:type ex:Cog17 eng:Part⟩ ⟩
```

(c) Corresponding triple model

$\forall a, b, i$ ⟨rdfs:subClassOf a b⟩
$\wedge$ ⟨rdf:type i a⟩
$\supset$ ⟨rdf:type i b⟩

(d) Rule implied by RDF-S semantics

```
⟨ ⟨rdf:type eng:Artifact rdfs:Class⟩
⟨rdf:type eng:Part rdfs:Class⟩
⟨rdfs:subClassOf eng:Part eng:Artifact⟩
⟨rdf:type ex:Cog17 eng:Part⟩
⟨rdf:type ex:Cog17 eng:Artifact⟩ ⟩
```

(e) Model derived from facts and rule

Fig. 2. The basic approach of DAMLJessKB.

on its ability to make interesting inferences, and consequently on its ability to accept valid inputs. DAMLJessKB avoids these problems by providing for such inferences. We see as one of its strengths the ability to add value to even simple Semantic Web applications by providing an easy to use DAML inference tool coupled with the excellent application building features of the underlying production system.

4 Approach

The DAMLJessKB mapping is illustrated in Figure 2. One of the basic properties of RDF is that no matter how complex its XML form, the underlying model can be seen simply as a list of triples. Each triple asserts a relation between a subject and an object through a predicate. DAMLJessKB maps these into facts in a production system.

Procedure. Given an XML source document (Figure 2(b)), an RDF parser generates a stream of triples (Figure 2(c)). These triples are asserted into the production system and rules derived from the semantics of the language are applied (Figure 2(d)) to populate the knowledge base with the additional facts which can be entailed from the input (Figure 2(e)). Additional methods are then invoked to carry out extra-logical aspects of the languages, such as loading ontologies from `daml:imports` statements, as well as tasks such as ontology debugging through looking for valid but probably unintended inputs.

The advantage of this approach is that information extracted and inferred from input documents is immediately available for use in the underlying production sytem. Such

```
<daml:Class rdf:about="#Piece" />

<daml:Class rdf:about="#Class">
 <daml:intersectionOf
   rdf:parseType="daml:collection">
  <rdfs:Class rdf:about="&daml;#Class" />
  <daml:Restriction>
   <daml:onProperty
     rdf:resource="&daml;#subClassOf" />
   <daml:hasValue rdf:resource="#Piece" />
  </daml:Restriction>
 </daml:intersectionOf>
</daml:Class>

<lego:Class rdf:about="#Brick" />

<daml:Class rdf:about="#Gear">
 <daml:subClassOf rdf:resource="#Piece" />
</daml:Class>

<lego:Brick rdf:about="#MyBrick" />
```

(a) A class of classes in DAML.

```
⟨ ⟨rdf:type lego:Brick daml:Class⟩
⟨rdfs:subClassOf lego:Brick lego:Piece⟩
⟨rdf:type lego:Gear lego:Class⟩
⟨rdf:type lego:MyBrick lego:Piece⟩ ⟩
```

(b) Relevant portion of corresponding inferred model.

Fig. 3. Example demonstrating a TBox/ABox intersection in DAML

systems are commonly used in knowledge-based applications and represent a powerful programming paradigm. DAMLJessKB allows developers to easily interwork the semantics of the input languages with their own application-specific reasoning.

Another advantage of this approach, as compared to many description logic implementations, is that it makes no distinction between the terminology of the ontologies being used and any instance data loaded. There is no "TBox/ABox" boundary in the description logic sense [14]. The syntax and semantics of RDF-S and DAML fully permit such reasoning—in fact, the language definitions make use of this sort of reasoning, for example in declaring `daml:Class` a subclass of `rdfs:Class`. A more complex example of reasoning on classes of classes is presented in Figure 3. In this example, DAMLJessKB correctly infers all entailments because terminological items are not treated specially, except that RDF-S and DAML symbols are present in the rulebase.

DAMLJessKB differs from most Semantic Web reasoners, which are built on description logic and general theorem provers, in that, by using a production system, it assumes a closed world (i.e., if a fact does not exist it is assumed to be false). This is in contrast to the RDF-S and DAML standards, which specify an open world assumption (i.e., all facts have to be proven either true or false). DAMLJessKB's closed world assumption has both positives and negatives. On the negative side, it makes some inferences suspect as they do not follow the exact semantics of the languages. On the positive side, closed world is often a reasonable assumption in practice: one can ask and answer many practical questions under the assumption that knowledge-bases used with DAMLJessKB will include all required data for making the needed inferences. If data is "unknown", the external application using DAMLJessKB can find out its value and assert if it is true or false. Indeed, we believe many real-world Semantic Web applications will need to make similar assumptions, albeit perhaps in different forms such as mechanically asserting closure axioms as a pre-reasoning step in reading input data—even if these are non-standard assumptions with respect to the language semantics. Indeed, closed world assumptions may prove necessary to enable wide acceptance and utility for Semantic Web applications.

5 Implementation

DAMLJessKB uses ARP ("Another RDF Parser") to load and parse XML documents containing RDF. ARP is part of the Jena toolkit [1] for parsing RDF documents, actively developed and supported by Hewlett Packard Labs. Amongst several available RDF parsers for Java, ARP was chosen because of its speed and support for DAML collection syntax. Unlike most parsers, ARP simply produces a stream of RDF triples from input triples. Other packages then make use of this stream, for example in the larger Jena package by constructing a graph model of the triples. This is ideal for use in DAMLJessKB as the only functionality required of the RDF parser is the generation of triples to be asserted into the production system. ARP supports `rdf:parseType="daml:collection"` syntactic constructs by creating the corresponding `daml:List` object and feeding it into the stream of triples.

As noted previously, DAMLJessKB uses Jess ("Java Expert System Shell") as its inference engine. Jess is actively developed and supported by Sandia National Laboratories. In many respects it can be seen as a Java version of CLIPS [15], the widely used C Language Interface Production System originally developed by NASA. The two share roughly the same architecture, expressiveness, and scripting capabilities. Jess was chosen over several other Java-based production systems because of its active development and support, tight interaction with Java programs, scripting language, and expressiveness. The Jess rule language includes elements not present in many other production systems, such as negation and arbitrary combinations of boolean conjunctions and disjunctions. We have found it necessary to use such capabilities in DAMLJessKB, for example using negation and conjunction in a rule to implement a form of universal quantification. Jess provides for excellent application development opportunities through utilizing features of the Java language and runtime environment to enable easy-to-use and powerful interaction with external programs. Its scripting language is also powerful enough to generate full applications entirely within the Jess system.

DAMLJessKB acts as a facade for the Jess object containing the knowledge base. The facade's primary purpose is to supply an interface for loading RDF documents into Jess and automatically applying RDF/RDF-S/XSD/DAML+OIL semantics. It also provides access to the knowledge base as if it were a normal Jess object while supplying additional general utility functionality, such as loading Jess files from Java archives (`jar` files) and alternate interfaces to the Jess query mechanism. The following subsections explain the process outlined in Section 4 in more detail.

5.1 Translation from Arp to Jess

Triples produced by ARP are collected and asserted into Jess. As the triples are collected, some simple manipulations on the stream are performed to handle syntactic translation, anonymous nodes, and datatypes. Most basic of these manipulations is translating URIs into valid Jess symbols by removing invalid characters. For example, tildes are replaced with an escape code. DAMLJessKB also inserts the dummy predicate `PropertyValue`

[1] Available at `http://www.hpl.hp.com/semweb/`

```
<daml:Datatype rdf:about="#AtLeast4">
  <xsd:minInclusive>
    <xsd:integer rdf:value="4" />
  </xsd:minInclusive>
</daml:Datatype>

<daml:Datatype rdf:about="#AtLeast6">
  <xsd:minInclusive>
    <xsd:integer rdf:value="6" />
  </xsd:minInclusive>
</daml:Datatype>
```

(a) Sample datatype definitions.

```
<lego:Brick rdf:about="#Brick1">
  <lego:length>12</lego:length>
</lego:Brick>

<lego:Brick rdf:about="#Brick2">
  <lego:length rdf:value="2" />
</lego:Brick>

<lego:Brick rdf:about="#Brick3">
  <lego:length><xsd:Integer rdf:value="8" />
  </lego:length>
</lego:Brick>
```

(b) Sample literal values.

Fig. 4. Example datatype definitions and literal values

in front of each triple, similar to the axiomatic DAML semantics [9]. This is necessary to enable variables and quantification on relations.

Anonymous objects present in the RDF XML are skolemized using a unique identifier generated by ARP. In addition, the unary predicate `anonymous` is asserted for each skolem symbol. In theory this could be used to incorporate some amount of reasoning about the existential nature of these nodes, as per the RDF specification. This marking has also proven useful in practice for doing such things as determining the relative importance of the object to the ontology designer or input source. For example, in generating class diagrams anonymous classes can be removed to reduce clutter as they are typically simply a means to the end of describing another class.

DAMLJessKB also looks for literal values as it collects triples and only allows them as objects of `rdf:value` statements. If they are encountered in other relations, a new `daml:Datatype` object is created and placed as the object of the statement. A new triple is then generated and incorporated into the stream stating that the `rdf:value` of the new datatype object is the literal which it replaced. This ensures a consistent `daml:Datatype` encapsulation for each literal without placing special demands on input ontologies or data. In addition, these literals remain compliant to the RDF and DAML standards such that triples containing literals can be reserialized from DAMLJessKB and used as input to another system. With literals treated in this consistent fashion it is possible to write rules implementing semantics for these datatype objects as well as other DAML elements containing literals, such as cardinality restrictions.

5.2 XML Schema Datatype Semantics

DAMLJessKB contains provisions for reasoning on literal values, implementing a form of `daml:Datatype` objects. These are based on Part Two of the XML Schema standard [5], Datatypes, using a non-standard RDF adaptation of XSD datatype definitions. Figure 4(a) shows some typical datatype definitions of this form while Figure 4(b) demonstrates some basic literal value syntax.

Figure 5(a) contains a rule for classifying literals as members of datatypes. From Figure 4(b), which shows CAD models for two Lego bricks, this rule will determine that `Brick3` and `Brick1` both have lengths of `AtLeast6` and `AtLeast4` in accordance with the two datatype definitions, as well as standard classes such as `xsd:positiveInteger`.

DAMLJessKB views datatypes as a class and therefore permits them to be used much as other classes might. For example, they can be combined through boolean oper-

```
(defrule mininclusive-classification
 (PropertyValue
  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  ?dt http://www.daml.org/2001/03/daml+oil#Datatype)
 (PropertyValue
  http://www.w3.org/2000/10/XMLSchema#minInclusive
  ?dt ?anon)
 (PropertyValue
  http://www.w3.org/1999/02/22-rdf-syntax-ns#value
  ?anon ?value)

 (PropertyValue
  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  ?inst
  http://www.w3.org/1999/02/22-rdf-syntax-ns#Literal)
 (PropertyValue
  http://www.w3.org/1999/02/22-rdf-syntax-ns#value
  ?inst ?ival)
 (test (and (integerp ?ival) (integerp ?value)
   (>= ?ival ?value)))

=> (assert
 (PropertyValue
  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
  ?inst ?dt)))
```

(a) Sample rule for classifying literals

```
(defrule mininclusive-subclassing
  (PropertyValue
   http://www.w3.org/1999/02/22-rdf-syntax-ns#type
   ?dt1 http://www.daml.org/2001/03/daml+oil#Datatype)
  (PropertyValue
   http://www.w3.org/2000/10/XMLSchema#minInclusive
   ?dt1 ?anon1)
  (PropertyValue
   http://www.w3.org/1999/02/22-rdf-syntax-ns#value
   ?anon1 ?value1)

  (PropertyValue
   http://www.w3.org/1999/02/22-rdf-syntax-ns#type
   ?dt2&~?dt1
   http://www.daml.org/2001/03/daml+oil#Datatype)
  (PropertyValue
   http://www.w3.org/2000/10/XMLSchema#minInclusive
   ?dt2 ?anon2)
  (PropertyValue
   http://www.w3.org/1999/02/22-rdf-syntax-ns#value
   ?anon2 ?value2)

  (test (and (integerp ?value1) (integerp ?value2)
    (>= ?value1 ?value2)))
=>
  (assert (PropertyValue
    http://www.w3.org/2000/01/rdf-schema#subClassOf
    ?dt1 ?dt2)))
```

(b) Sample rule for terminological reasoning on datatypes

Fig. 5. Examples of DAMLJessKB's treatment of literals and datatypes

```
(defrule subclass-instances
  (PropertyValue
    http://www.w3.org/2000/01/rdf-schema#subClassOf
    ?child ?parent)
  (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    ?instance ?child)
=>
 (assert
  (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    ?instance ?parent)))
```

(a) Rule implementing basic notion of subclassing.

```
(defrule rdfs-domain
  (declare (salience -100))
  (PropertyValue
    http://www.w3.org/2000/01/rdf-schema#domain
    ?p ?c)
  (PropertyValue ?p ?i ?o)
  (not (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    ?i ?c))
=>
  (assert (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    ?i ?c))
  (gentle-warning "Set object '" ?i "' type to '" ?c
    "' due to a domain restriction on '" ?p "'"))
```

(b) One rule corresponding to semantics of `rdfs:domain`.

Fig. 6. Examples of DAMLJessKB's instance reasoning

ations to create new classes. In addition, special subsumption relationships are defined based on the particular semantics of the datatypes. This is very similar to the use of `daml:Restriction` classes, discussed later. Figure 5(b) demonstrates a rule implementing subsumption between `minInclusive` constraints. Given the two definitions in Figure 4(a), this rule will assert that `AtLeast6` is a subclass of `AtLeast4`.

5.3 Instance Data Reasoning

The set of rules implementing DAMLJessKB's reasoning can be roughly seen as falling into two categories. One concerns reasoning on instances of classes. The second concerns terminological reasoning, determining relationships between the classes themselves. This is roughly analogous to ABox and TBox reasoning in description logic systems.

Two examples have already been shown in regards to XSD semantics. The rule shown in Figure 5(a) can be seen as falling into the former, determining the relationships between instances and classes. Figure 5(b) contains a rule which corresponds to the latter category of rules, determining relationships between classes.

One of the most basic elements of the RDF-S and DAML languages are the `rdfs:subClassOf` and `daml:subClassOf` statements, which are defined to be equivalent. These properties are used to specify a subclass relationship between two classes. One of the intuitive notions of this relation is that any instance of a subclass is an instance of the parent class. Figure 6(a) implements this portion of the semantics of the relation, a very basic inference to perform on instances. Despite its simplicity, even this basic inference is extremely useful in practice and adds a great of capability to any application reading in data encoded according to a DAML or RDF-S ontology. For example, this rule provides for inferring that an object explicitly stated to be of a class in a taxonomy is also an instance of all the ancestors of that class.

One of the benefits of having formal semantics for DAML is that interpretations and assumptions of the language elements can be identified, characterized, and validated. Such assumptions can often be seen in practice, especially when the language is used by developers more accustomed to traditional programming than knowledge representation. A small but interesting example of this concerns `rdfs:domain` statements.

This property is used to assign a domain to another property, specifying the constraint that only instances of the domain class may have values for this property. Nearly all programmers are familiar with type-checking and this constraint is easily interpreted as a form of this. Many would make the assumption, based on their familiarity with traditional programming languages, that if an object has been asserted to have a value for such a constrained property but is not a member of the domain, then there is an error.

However, the actual inference in such a situation is that if an object, not otherwise of the domain, has been explicitly asserted to have a value for the constrained property, then it is indeed a member of the domain. Figure 6(b) contains a rule corresponding to this inference entailed by the semantics of the `rdfs:domain` property. The rule implements the actual semantics of the language element, however, recognizing that there is a common assumption about this property it also generates a warning. This warning indicates that such a situation exists and provides for some use of the domain constraint as a basic consistency checking device.

We note that such inconsistencies are formally characterizable using DAML. For example, if an instance of a class was found to have a value for a property constrained to a domain disjoint with the class, an actual inconsistency would be raised because the object would be a member of two mutually exclusive classes. However, at least currently, many DAML ontologies are not defined formally and exhaustively enough to contain such information. Many DAML ontologies contain an assumption on the part of the developers that classes are disjoint unless stated or reasoned to be otherwise. Therefore, some special actions such as the warning generated by the rule in Figure 6(b) are warranted in practice to help identify such assumptions.

```
<daml:Class rdf:about="#RobotLabMotor">
 <daml:intersectionOf
  rdf:parseType="daml:collection">
   <daml:Class rdf:about="&lego;#Piece" />
   <daml:Class rdf:about="&artifact;#Motor" />
   <daml:Class rdf:about="&robotlab;#KitItem" />
 </daml:intersectionOf>
</daml:Class>
```

(a) DAML snippet defining an intersection of classes

```
(defrule intersection-of-subsumption
 (declare (salience -50))

 (PropertyValue
  http://www.daml.org/2001/03/daml+oil#intersectionOf
  ?topClass ?topList)
 (PropertyValue
  http://www.daml.org/2001/03/daml+oil#intersectionOf
  ?botClass&~?topClass ?botList)

 (not (and (list-item ?topList ?y)
  (not (or (list-item ?botList ?y)
   (and (list-item ?botList ?x)
    (PropertyValue
     http://www.w3.org/2000/01/rdf-schema#subClassOf
     ?x ?y))))))

=>
  (assert (PropertyValue
    http://www.daml.org/2001/03/daml+oil#subClassOf
    ?botClass ?topClass)))
```

(b) Rule implementing subsumption of class intersections

```
<daml:Restriction>
  <daml:onProperty rdf:resource="#input" />
  <daml:cardinalityQ rdf:value="1" />
  <daml:hasClassQ
    rdf:resource="&flow;#ElectricalFlow" />
</daml:Restriction>
```

(c) DAML snippet defining a cardinality constraint

```
(defrule mincardinalityq-subsumption
  (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    ?restriction1
    http://www.daml.org/2001/03/daml+oil#Restriction)
  (PropertyValue
    http://www.daml.org/2001/03/daml+oil#onProperty
    ?restriction1 ?prop1)
  (PropertyValue
    http://www.daml.org/2001/03/daml+oil#minCardinalityQ
    ?restriction1 ?lit1)
  (PropertyValue
    http://www.daml.org/2001/03/daml+oil#hasClassQ
    ?restriction1 ?class1)
  (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#value
    ?lit1 ?val1)

  (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    ?restriction2&~?restriction1
    http://www.daml.org/2001/03/daml+oil#Restriction)
  (PropertyValue
    http://www.daml.org/2001/03/daml+oil#onProperty
    ?restriction2 ?prop2)
  (or (PropertyValue
    http://www.daml.org/2001/03/daml+oil#minCardinalityQ
    ?restriction2 ?lit2)
    (PropertyValue
      http://www.daml.org/2001/03/daml+oil#cardinalityQ
      ?restriction2 ?lit2))
  (PropertyValue
    http://www.daml.org/2001/03/daml+oil#hasClassQ
    ?restriction2 ?class2)
  (PropertyValue
    http://www.w3.org/1999/02/22-rdf-syntax-ns#value
    ?lit2 ?val2)

  (test (and (integerp ?val1) (integerp ?val2)
    (>= ?val2 ?val1)))
  (or (test (eq ?class2 ?class1))
    (PropertyValue
      http://www.w3.org/2000/01/rdf-schema#subClassOf
      ?class2 ?class1))
=>
  (assert (PropertyValue
    http://www.daml.org/2001/03/daml+oil#subClassOf
    ?restriction2 ?restriction1)))
```

(d) Rule implementing subsumption between cardinality restrictions

Fig. 7. Examples of DAMLJessKB's terminological reasoning

5.4 Terminological Reasoning

Utilizing the full power of the Semantic Web requires inference on the relationships between classes—terminological reasoning. Through the semantics of description logic, objects and classes can be automatically compared, contrasted, and otherwise reasoned on. In order to do so the classes of objects present in the ontology have to be defined using a description logic language, in this case DAML.

One of the most commmon elements in such descriptions is the definition of a class as the intersection of a set of classes—conjunction. Figure 7(a) contains a snippet using the `daml:intersectionOf` class expression to define such an intersection. In the snippet, the class `RobotLabMotor` is declared as consisting of those objects which are members of the classes `lego:Piece`, `artifact:Motor`, and `robotlab:KitItem`, which are either primitive terms or defined elsewhere.

A very common terminological inference is subsumption between such intersections. Figure 7(b) shows an element of DAMLJessKB's implementation of such reasoning. The rule determines subclass relationships between intersections of classes by implementing structural subsumption on classes consisting solely of such intersections. Intuitively, the rule implements the idea that a class composed of the intersection of a set of classes is a subclass of a class composed of the intersection of a subset of those classes or subclasses of those classes. In the case where the two intersections are equivalent, each class will be asserted as a subclass of the other. Note that for convenience the set of classes in the intersection are held in a closed world list form which corresponds to the open world `daml:List` object actually created by the RDF parser.

Another common descriptive element is to declare the cardinality of properties for classes of objects. DAML contains several mechanisms for asserting such constraints. Figure 7(c) demonstrates an anoymous class declared to be subject to the constraint that it has one value for the `input` property of type `flow:ElectricalFlow`, although instances of this class may have other values of other types for that property. `daml:Restriction` objects are DAML classes associated with various types of constraints, including cardinality and type qualification. These are expressed through properties such as `daml:cardinalityQ` to indicate a qualified cardinality and `daml:onProperty` which declares the property on which the restriction is being placed.

Figure 7(d) shows a rule implementing subsumption reasoning between qualified cardinality restrictions. This corresponds to the intuitive notion that a class declared to have more fillers for a given property of a given class is a subclass of of a class declared to have less fillers for the given property of the same class or a super-class of the type qualification. The inferences provided by the semantics for these cardinality restrictions and class intersections will be used extensively in the examples presented in Section 6.

6 Using DAMLJessKB

This section gives two basic demonstrations of DAMLJessKB. The first is a simple example of a tool which would be crippled without being able to reason on DAML semantics. Closer to the actual application areas in which we are working, the second example shows a variational engineering design problem to which we hope to apply DAML and the Semantic Web.

Simple Example: Inferring Subclasses. Tools for creating Semantic Web content require reasoning capabilities just as applications do. Whether authoring ontologies or modeling information, reasoning is necessary to fully support the user with such tasks as consistency checking, graphical displays, and navigation. This is why ontology authoring environments such as Ontolingua and Protegé incorporate reasoning mechanisms. DAMLJessKB is well positioned to enable such reasoning for even small tools. Such tools include ontology markup cross referencers, class graphers, and basic information extracters. A number of these tools have been developed for DAML and a list is maintained at `http://www.daml.org/tools/`. However, many of these tools include no or very simplistic reasoning and therefore can not accept many valid inputs or produce inaccurate results. DAMLJessKB provides one option for easily incorporating Semantic Web reasoning into such tools.

```
(defquery direct-subclasses
  (PropertyValue
    http://www.daml.org/2001/03/daml+oil#subClassOf
    ?y&~:(guess-standard ?y)
    ?x&~?y&~:(guess-standard ?x))
  (not (anonymous ?y)) (not (anonymous ?x))

  (not (and
    (PropertyValue
      http://www.daml.org/2001/03/daml+oil#subClassOf
      ?y ?z&~?x&~?y)
    (PropertyValue
      http://www.daml.org/2001/03/daml+oil#subClassOf
      ?z ?x)
    (not (or (anonymous ?z)
      (PropertyValue
        http://www.daml.org/2001/03/daml+oil#subClassOf
        ?z ?y)))))
)

(printout t "digraph g {" crlf crlf
  "node [shape=box,fontname=Helvetica,fontsize=10];"
  crlf "edge [dir=back];" crlf crlf)

(bind ?res (run-query direct-subclasses))
(while (call ?res hasNext)
  (bind ?triple (call (call (call ?res next) fact 1)
    get 0))
  (bind ?obj (nth$ 3 ?triple))
  (bind ?subj (nth$ 2 ?triple))

  (printout t "    " (guess-name ?obj) " -> "
    (guess-name ?subj) ";" crlf))

(printout t crlf "}" crlf)
```

(a) Jess program generating graph for `dot`

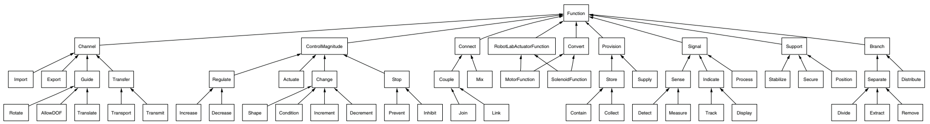
```
<daml:Class rdf:about="&func;#Function">
  <daml:disjointWith rdf:resource="&flow;#Flow" />
  <daml:disjointWith rdf:resource="&artifact;#Artifact" />
</daml:Class>

<daml:Class rdf:about="&func;#Convert">
  <daml:subClassOf rdf:resource="&func;#Function" />
  <daml:disjointWith rdf:resource="&func;#Branch" />
  <daml:disjointWith rdf:resource="&func;#Channel" />
  <daml:disjointWith rdf:resource="&func;#Connect" />
  <daml:disjointWith
    rdf:resource="&func;#ControlMagnitude" />
  <daml:disjointWith rdf:resource="&func;#Provision" />
  <daml:disjointWith rdf:resource="&func;#Signal" />
  <daml:disjointWith rdf:resource="&func;#Support" />
</daml:Class>

<daml:Class rdf:about="&func;#MotorFunction">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="&func;#Convert" />
    <daml:Restriction>
      <daml:onProperty rdf:resource="#input" />
      <daml:toClass
        rdf:resource="&flow;#ElectricalFlow" />
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#output" />
      <daml:toClass rdf:resource="&flow;#Rotation" />
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:about="&func;#RobotLabActuatorFunction">
  <daml:subClassOf rdf:resource="&func;#Function" />
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="&func;#MotorFunction" />
    <daml:Class rdf:about="&func;#SolenoidFunction" />
  </daml:unionOf>
</daml:Class>
```

(b) Snippet from an input ontology

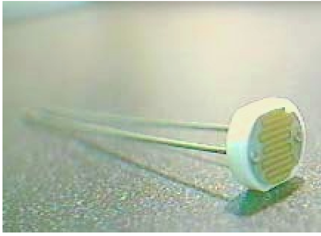


(c) Generated graph

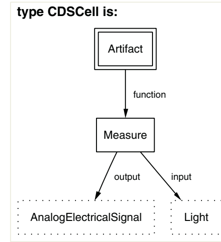
Fig. 8. Simple application to generate subclass graph

For example, generating subclass graphs is a very simple capability which can greatly aid ontology authors and readers. Several programs and web services exist to generate such graphs, however most operate solely on explicit RDFS or DAML `subClassOf` statements. In order to accurately display all the subclass relationships and accept all valid ontology inputs (including such examples as that in Figure 3), the tool must be able to reason on the inputs. Figure 8 shows a simple version of such a tool which utilizes DAMLJessKB. Figure 8(a) contains a small program in Jess' scripting language for generating the subclass graph by generating output suitable for use with `dot`, an automatic graph layout tool[2]. The core of the program is a query for all the subclass relationships which have been inferred. Figure 8(b) contains a small portion of an input ontology and Figure 8(c) the generated graph.

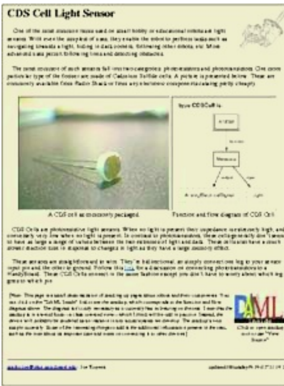
[2] Available at `http://www.research.att.com/sw/tools/graphviz/`



(a) A Cadmium Sulfide (CDS) cell



(b) CDS cell function and flow diagram



(c) Webpage for CDS cell

```
<daml:Class rdf:about="#Sensor">
 <daml:intersectionOf
   rdf:parseType="daml:collection">
  <daml:Class rdf:about="&eng;#Artifact" />

  <daml:Restriction>
   <daml:onProperty rdf:resource="&eng;#function"/>
   <daml:minCardinalityQ rdf:value="1" />

   <daml:hasClassQ>
    <daml:Class>
     <daml:intersectionOf
       rdf:parseType="daml:collection">
      <daml:Class rdf:about="&func;#Measure" />

      <daml:Restriction>
       <daml:onProperty
         rdf:resource="&eng;#output" />
       <daml:minCardinalityQ rdf:value="1" />
       <daml:hasClassQ
         rdf:resource="&flow;#MeasureSignal" />
      </daml:Restriction>

      <daml:Restriction>
       <daml:onProperty rdf:resource="&eng;#input"/>
       <daml:minCardinalityQ rdf:value="1" />
       <daml:hasClassQ rdf:resource="&flow;#Flow"/>
      </daml:Restriction>

     </daml:intersectionOf>
    </daml:Class>
   </daml:hasClassQ>
  </daml:Restriction>
 </daml:intersectionOf>
</daml:Class>
```

(d) DAML description of sensor class

Fig. 9. Artifact function and flow modeling and a comparison of two assembly models

Real-World Application: Variational Engineering Design. As outlined in Section 2, we are interested in reasoning about electromechanical assemblies and components. This includes tasks such as determining if an artifact performs a given function and searching for artifacts in design repositories to find similarities.

Consider the problem of designing LEGO robots, such as with the very popular Lego Mindstorms robot kits[3]. Figure 9(a) shows a typical light sensor component used in Lego robot kits; Figure 9(b) shows an engineering function and flow diagram [16,17] for this component. Function and flow diagrams provide an abstract representation of an assembly, its components and their intended behavior. This diagram notes that the sensor measures an input light source and outputs an electrical signal as the measure. However,

[3] http://mindstorms.lego.com/

function-flow models are typically lacking in formal semantics. Therefore we cannot automate such tasks as comparing these diagrams or searching large online repositories for functional patterns.

We have developed a formalization of these represenations by attributing description logic semantics to the diagrams. For example, the function and flow representation of the CDS sensor presented in Figure 9(b) can be mechanically interpreted as corresponding to the following description logic statement:

$$\text{CDSCell} \equiv (\text{Artifact} \wedge \exists \text{function}.(\text{Measure} \wedge \exists \text{output}.\text{AnalogElectricalSignal} \wedge \exists \text{input}.\text{Light})).$$

The use of DAML as the description logic language enables us to embed these formal representations into Web content or into an XML-based database schema. Figure 9(c) shows a web page used to provide information about these sensors to students in an introductory robotics class at Drexel University. By including DAML markup of such function and flow diagrams in Web pages we can perform such tasks as searching for components to perform specific functions. To illustrate this, Figure 9(d) contains a DAML version of a similar function and flow diagram for a general sensor. The component described by the webpage in Figure 9(c) can be compared to this class by loading both into the DAMLJessKB knowledge base and querying for relationships between the two. In this case the component is correctly incorporated into the terminology as a subclass of the general sensor; the software has determined that the component is a sensor without that information being explicitly stated.

7 Conclusions

Practical tools are needed for the vision of the Semantic Web to become fully realized. This paper introduced DAMLJessKB, which the authors believe is one such tool. With DAMLJessKB, users can perform inference based on semantics of the description logic which forms the basis of DAML. As more and more websites, network services, databases, and knowledge-bases look to DAML as a de facto representation syntax, DAMLJessKB will become one in a suite of tools that allow users to truly leverage the new-found shared semantics. In this way, we are hoping to contribute toward full reasoning with the Semantic Web.

DAMLJessKB is being actively used in a number of academic and government research projects. We have made DAMLJessKB publicly available under the GNU General Public License at
`http://edge.mcs.drexel.edu/assemblies/software/damljesskb/`.
We hope that this article broadens interest in DAMLJessKB and helps to create and grow a community of users who are working to improve it.

Acknowledgements. This work was supported in part by National Science Foundation (NSF) Knowledge and Distributed Intelligence in the Information Age (KDI) Initiative Grant CISE/IIS-9873005; Office of Naval Research Grant N00014-01-1-0618 and the National Institute of Standards and Technology (NIST) Grant 70-NAN-B2H0029. Any

opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Office of Naval Research, or other supporting organizations.

References

1. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American (2001)
2. Hendler, J.: Agents and the semantic web. IEEE Intelligent Systems (2001)
3. W3C: Resource Description Framework (RDF) model and syntax specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (1999)
4. W3C: Resource Description Framework Schema Specification (RDF-S). http://www.w3c.org/TR/2000/CR-rdf-schema-20000327/ (2000)
5. World Wide Web Consortium: XML Schema Part 2: Datatypes (XSD). http://www.w3.org/TR/xmlschema-2/ (2001)
6. DARPA: DAML march 2001 specifications (DAML+OIL). http://www.daml.org/2001/03/daml+oil-index (2001)
7. Friedman-Hill, E.: Jess: The rule engine for the Java platform. herzberg.ca.sandia.gov/jess/ (1995)
8. Shah, U., Finin, T., Joshi, A., Mayfield, J., Cost, R.: Information retrieval on the semantic web. In: ACM Conference on Information and Knowledge Management. (2002)
9. Fikes, R., McGuinness, D.L.: An axiomatic semantics for RDF, RDF Schema, and DAML+OIL. Technical Report KSL-01-01, Knowledge Systems Laboratory, Stanford University (2001)
10. van Harmelen, F., Patel-Schneider, P.F., Horrocks, I.: A model-theoretic semantics for DAML+OIL. http://www.daml.org/2001/03/model-theoretic-semantics (2001)
11. Szykman, S., Bochenek, C., Racz, J.W., Senfaute, J., Sriram, R.D.: Design repositories: Engineering design's new knowledge base. IEEE Intelligent Systems **15** (2000) 48–55
12. Szykman, S., Sriram, R.D., Regli, W.C.: The role of knowledge in next-generation product development systems. ASME Transactions, the Journal of Computer and Information Science in Engineering **1** (2001) 3–11
13. Horrocks, I.: The FaCT system. http://www.cs.man.ac.uk/~horrocks/FaCT/ (1999)
14. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook. Cambridge University Press (2002)
15. NASA: Clips: A tool for building expert systems. http://www.ghgcorp.com/clips/CLIPS.html (2002)
16. Szykman, S., Racz, J.W., Sriram, R.D.: The representation of function in computer-based design. In: ASME Design Engineering Technical Conferences, 11th International Conference on Design Theory and Methodology, New York, NY, USA, ASME, ASME Press (1999) DETC99/DTM-8742.
17. Pahl, G., Beitz, W.: Engineering Design – A Systematic Approach. 2nd edn. Springer, London, UK (1996)