International Journal of Semantic Computing
Vol. 7, No. 3 (2013) 215–236

DOI: 10.1142/S1793351X13500025

World Scientific
www.worldscientific.com

TOWARD A UNIFIED ENGLISH-LIKE REPRESENTATION OF SEMANTIC MODELS, DATA, AND GRAPH PATTERNS FOR SUBJECT MATTER EXPERTS

ANDREW CRAPO* and ABHA MOITRA†

GE Global Research
Niskayuna, NY 12309, USA
**crapo@research.ge.com*
†*abha.moitra@research.ge.com*

The Semantic Application Design Language (SADL) combines advances in standardized declarative modeling languages based on formal logic with advances in domain-specific language (DSL) development environments to create a controlled-English language that translates directly into the Web Ontology Language (OWL), the SPARQL graph query language, and a compatible if/then rule language. Models in the SADL language can be authored, tested, and maintained in an Eclipse-based integrated development environment (IDE). This environment offers semantic highlighting, statement completion, expression templates, hyperlinking of concepts to their definition, model validation, automatic error correction, and other advanced authoring features to enhance the ease and productivity of the modeling environment. In addition, the SADL language offers the ability to build in validation tests and test suites that can be used for regression testing. Through common Eclipse functionality, the models can be easily placed under source code control, versioned, and managed throughout the life of the model. Differences between versions can be compared side-by-side. Finally, the SADL-IDE offers an explanation capability that is useful in understanding what was inferred by the reasoner/rule engine and why those conclusions were reached. Perhaps more importantly, explanation is available of why an expected inference failed to occur. The objective of the language and the IDE is to enable domain experts to play a more active and productive role in capturing their knowledge and making it available as computable artifacts useful for automation where appropriate and for decision support systems in applications that benefit from a collaborative human-computer approach. SADL is built entirely on open source code and most of SADL is itself released to open source. This paper explores the concepts behind the language and provides details and examples of the authoring and model lifecycle support facilities.

Keywords: Controlled English; graph pattern; ontology; OWL; semantic model.

1. Introduction

"Model" means different things to different people. One interesting use of the word is in the phrase "mental model". People are arguably pretty good at organizing their observations and perceptions of the world into models in their minds that allow them

to explain the past and the present and predict the future. It's how we survive and are able to thrive. Creating and revising mental models seems to happen very naturally and usually without a lot of conscious effort — it is what we do with our over-sized brains!

It's often a different matter when we try to communicate our mental models to others. It usually takes a substantial amount of effort on the part of both the giver and the receiver for one person's model to be understood by another. The manner of communication normally requiring the least amount of conscious effort is verbal communication. That part of our brain is well exercised. But if we want to persist the externalization of our mental model in an artifact of some kind, the effort usually goes up substantially. Of course we can just record the audio or video of our explanations of our thinking, but if you've tried to do that in a way that you are satisfied with in the end you have probably found it required considerable effort. For millennia the standard way of preserving our models has been through writing. Writing can be hard work indeed.

The next level of difficulty, relatively new to the human experience, is to capture our models in computable artifacts so that machines can use our models to draw the same conclusions that we would have drawn, given a particular scenario. There are a variety of ways of doing this ranging from writing procedural programs to creating spreadsheets to creating declarative models in a formal logic language. This paper will explore ways to enhance our ability to accomplish the latter with less effort and with results that are more maintainable, leading to longer useful model life. While there are a variety of logical, declarative modeling languages such as Prolog and the Common Logic family, we will use the Web Ontology Language (OWL) [15] as an example, along with its companion query language SPARQL [20] and a representative rule language.

The purpose of the work reported in this paper is to make it easier and more effective for subject matter experts (SMEs), also known as domain experts, to capture knowledge in a logic-based and if/then rule-based modeling environment (OWL + SPARQL + rules). We will identify and illustrate a number of approaches that we have used in striving to achieve this goal. Some of these solutions are embodied in an English-like (controlled-English) language called the Semantic Application Design Language (SADL) [18]. Other solutions are implemented in an Eclipse-based integrated development environment (IDE) called the SADL-IDE. In discussing each significant aspect of our approach we will discuss the concept, illustrate model implementations where appropriate in OWL, and then illustrate how we have improved things for the modeler using SADL and/or the SADL-IDE. We will begin with the underpinnings of declarative, graph-based modeling and the design principles underlying SADL in Secs. 2 and 3. In Sec. 4 we will delve into the capabilities of the IDE. In Sec. 5 we will briefly mention and compare SADL to other controlled-English languages. Finally in Sec. 6 we will briefly describe three different applications of SADL in real-world projects.

2. The World (in Our Head) Is a Graph

To set the stage let us talk about the nature of models. It is theorized that people's mental models are associative in nature and can be represented as directed graphs [7– 9]. Many of the nodes in our mental graph correspond to abstractions — patterns that we have identified and to which we have given names. One kind of abstraction is what we often call types or classes. Formalizations of this abstraction are often based on set theory. Examples of classes are trees, rocks, people, cars, languages, etc. Another kind of abstraction forms the edges in our graph. These are types of associations, relationships, or characteristics (attributes), all of which we will call properties, that we "see" in the world. Examples of properties include friend, spouse, child, employer, owner, author, age, height, weight, etc. Graph edges are visible in Fig. 1.

Class abstractions can further be divided into classes whose members are defined by their intrinsic characteristics and classes whose members are defined by their relationship to other things. For example, a person can be further classified as a man or a woman based on intrinsic characteristics — the classification can be performed by looking at the thing itself. On the other hand, a person can be classified as a mother or a friend only by looking at relationships between the thing and other things. These two ways of classifying are what Sowa calls "firstness" and "secondness", attributing the concepts and the names to Charles Peirce [19]. Secondness is also often called "roles". A role is a classification based on participation in a relationship.

Let us start with the very simple graph shown in Fig. 1. Since OWL has no innate temporal modeling capability, we will ignore time and all English representations will be in present tense.

It is interesting to consider the various ways that one might represent these constructs in the English language. Such a text might go something like this. "The man Abraham is a US president. He is married to a woman named Mary. They have a child named Robert who's birthdate is August 1, 1843."

Now let us represent the same information as an OWL snippet. We will use the RDF N3 format [14] in Fig. 2 as it is a more terse representation than RDF/XML or

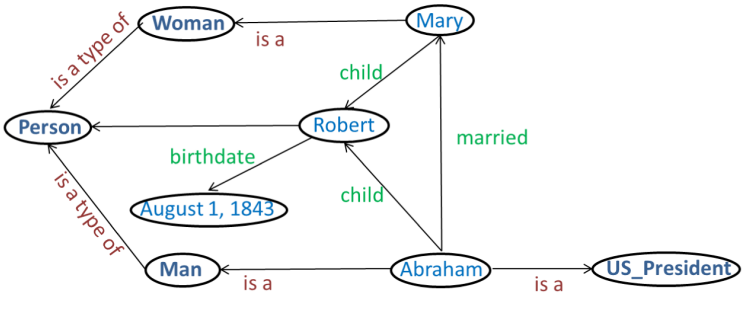

Fig. 1. A graph of some genealogical information.

```
lincolns:Robert
    a gen:Person ;
    gen:birthdate "1843-08-01"^^xsd:date .
lincolns:Mary
    a gen:Woman ;
    gen:child lincolns:Robert .
lincolns:Abraham
    a [ a owl:Class ;
      owl:intersectionOf (gen:Man gen:US_President)
      ] ;
    gen:child lincolns:Robert ;
    gen:married lincolns:Mary .
```

Fig. 2. Our simple graph in OWL RDF N3 format.

N-Triple and actually has some of the same kinds of compression of graph triples as does SADL — as explained later in this paper.

As our first introduction to SADL, we represent the same information as a SADL snippet in Fig. 3. Note that class names are in dark blue bold, property names are in green bold, instances (individuals) are in light blue, and literals have a grey background. Keywords in the language are in maroon, although keywords may also be used as non-keywords by escaping them with a preceding "^" (caret). Concept (class, property, instance) names cannot have spaces but can use underscores or hyphens. Names must start with an alpha character, not with a number, to comply with XML URI requirements. In our examples we will use the OWL convention of capitalizing class and instance names and not capitalizing property names, but there is no requirement to do so. The language is case-sensitive; "child" and "Child" are not the same concept. As we noted above, this is actually useful in trying to mimic English constructs where the role played ("Child") is a noun (class) but the predicate ("child") identifies an association. There are multiple ways that this text could be structured in SADL but each renders the same OWL model — the one shown in Fig. 3.

The bit of English text given in the three sentences following Fig. 1 is deceptively simple because it uses many concepts that each of us already has defined, more or less similarly, in our own mental models. To make this small model into a computable artifact, "understandable" by machines as well as people, these concept definitions (the t-box in semantic modeling terms) must be explicit and available to

```
Abraham is a{Man and US_President},
    married (a Woman Mary with child (a Person Robert
                                      with birthdate "August 1, 1843")),
    has child Robert.
```

Fig. 3. Our simple graph in SADL.

anyone/anything reading this instance data model (a-box in semantic modeling terms). Figures 4 and 5 show the concept definitions used in Figs. 2 and 3 in OWL and SADL, respectively. These figures include the header information where namespaces and prefixes are defined.

Let us note some of the constructs of the SADL language that make it more like natural language and/or easier for SMEs to understand.

(1) Optional qualified name. The identifiers of named concepts in OWL/RDF are URIs. A URI consists of a namespace and a local name separated by a "#" symbol. A prefix can be defined as an alias for a namespace, allowing named concepts to be expressed unambiguously as qualified names — a prefix followed by a local name, separated by a colon as illustrated in Figs. 2 and 4,

```
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix gen:     <http://sadl.org/TestSadlIde/geneology#> .
@prefix owl:     <http://www.w3.org/2002/07/owl# > .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
rdfs:comment     "This ontology was created from a SADL file 'genealogy.sadl'
            and should not be edited."@en ;
owl:versionInfo  "$Revision:$ Last modified on   $Date:$" .
<> a owl:Ontology ;
gen:Male a gen:Gender .
gen:Female a gen:Gender .
gen:Gender a owl:Class ; owl:equivalentClass [ a owl:Class ; owl:oneOf ( gen:Male gen:Female ) ] .
gen:Person a owl:Class ;
  rdfs:subClassOf [ a owl:Restriction ; owl:maxCardinality "1"^^xsd:int ; owl:onProperty gen:gender ];
  rdfs:subClassOf [ a owl:Restriction; owl:maxCardinality "1"^^xsd:int ; owl:onProperty gen:birthdate] .
gen:birthdate a owl:DatatypeProperty ; rdfs:domain  gen:Person ; rdfs:range xsd:date .
gen:gender  a owl:ObjectProperty ; rdfs:domain  gen:Person ; rdfs:range gen:Gender .
gen:child a owl:ObjectProperty ; rdfs:domain  gen:Person ; rdfs:range gen:Person .
gen:married  a owl:ObjectProperty ; rdfs:domain  gen:Person ; rdfs:range gen:Person .
gen:Man a owl:Class ;
  rdfs:subClassOf gen:Person ;
  owl:equivalentClass [ a owl:Class ; owl:intersectionOf ( gen:Person [ a owl:Restriction ;
                                                                        owl:hasValue gen:Male ;
                                                                        owl:onPropertygen:gender
                                                                        ] ) ] .
gen:Woman a owl:Class ;
  rdfs:subClassOf gen:Person ;
  owl:equivalentClass [ a owl:Class ; owl:intersectionOf ( gen:Person [ a owl:Restriction ;
                                                                        owl:hasValue gen:Female ;
                                                                        owl:onProperty gen:gender
                                                                        ] ) ] .
gen:Parent a owl:Class ;
  rdfs:subClassOf gen:Person ;
  owl:equivalentClass [ a owl:Class ; owl:intersectionOf ( gen:Person [ a owl:Restriction ;
                                                                        owl:minCardinality "1"^^xsd:int ;
                                                                    owl:onProperty gen:child
                                                                        ] ) ] .
gen:US_President a owl:Class ; rdfs:subClassOf gen:Person .
```

Fig. 4. Concept definitions in OWL RDF N3 format.

```
uri "http://sadl.org/TestSadlIde/genealogy"
alias gen version"$Revision:$ Last modified on  $Date:$".

Person is a class,
described by birthdate with a single value of type date,
described by gender with a single value of type Gender,
described by child with values of type Person,
described by married with values of type Person.

Gender is a class, must be one of {Male, Female}.

A Person is a Man only if gender always has value Male.
A Person is a Woman only if gender always has value Female.
A Person is a Parent only if child has at least 1 value.
```

Fig. 5. Concept definitions in SADL.

e.g. "owl:Class". In many usages concept names are unique within the set of namespaces referenced by a model. SADL allows the use of qualified names but does not require it unless the local name is not unique in the context. This makes the SADL statements more readable and the translator is responsible for creating qualified names and complete URIs in the generated RDF syntax. (The content of Figs. 2 and 4 was generated automatically by SADL.)

(2) Compound statements. It is not necessary to repeat the subject for each RDF statement (subject → predicate → object). For example, the subject Abraham in Fig. 3, line 1, is the subject of three statements. Note that the RDF N3 syntax, unlike RDF/XML and N-Triple syntax, also allows the same kind of multiple statements with a single subject as shown in Fig. 2.

(3) Filler words. There are optional filler words in SADL that make the phrases more English-like. For example, the phrase "Mary has child Robert" could also have been written as the raw triple "Mary child Robert" but the "has" makes it sound more natural. (Note that we do not allow "Mary has a child Robert" because that form is actually using "child" as a role class ("a child" expresses a classification of "Robert"), not as a predicate, and the predicate in this phrase is the ambiguous "has".) However, "has" does not always sound natural, depending on the type of the English verb equivalent. For example, the phrase "Abraham married Mary", which uses a transitive action verb, does not need a filler. In compound statements the filler word "with" instead of "has" sometimes makes the phrase more natural, as in "... with birthdate ...".

(4) Embedded definitions. The embedding of new definitions in parenthetical expressions is a compromise. Normally in English we might say something like "Abraham married a woman named Mary." It is implicit in this sentence that we have not previously encountered Mary and we are being told, since it is our first encounter, that she is a woman. The parentheses cue the SADL parser that this is a new definition.

In summary, associative networks are graphs and graph patterns are what we perceive and what we communicate about. Natural language is a graph language with many different forms and many shortcuts to reduce verbosity, many of which introduce ambiguity. With some significant restrictions, natural language (e.g. English) can be an unambiguous yet powerful representation capturing the knowledge of a domain as a semantic model.

3. More about Formal, Graph-Based Models and How They Differ from Informal Models

There are two important observations to note about graph-based models, observable in our natural language, that must be taken into account in constructing a formal modeling language. One has to do with how we identify things and the second has to do with how we express patterns useful in asking questions or expressing conditional statements, the natural-language equivalent of rules.

3.1. *Identity by association*

Most things in our models of the world do not have unique identifiers — they are identified by association with those things which do have identifiers. For big things, we create identifiers to make them easier to track. An automobile has a vehicle identification number (VIN). Most jurisdictions require that it have a license plate affixed to make its identity visible from some distance. The smaller things, however, have no unique identity other than the relationships that provide the context of their existence. Hence I might refer to "my eye glasses" or "my computer glasses" or "my pen" or "my shirt". Even when there is identity it is not necessarily easy to use that identity for everyday purposes. Hence we say "Please come to dinner at my house." Or, "Can we drive your car to the restaurant?" "My computer" is usually understood to be the one in my possession. It is our experience that any modeling system that requires that the modeler provide a unique identity for each thing modeled will be found to be very cumbersome to use.

In semantic technology terms, things which are not named are represented by blank nodes or bnodes. These nodes can be retrieved from the graph by their associations. In the example of Fig. 2 we saw that we can define an instance in-line or parenthetically. In Fig. 2 the parenthetically defined instances have names but SADL does not require this. In SADL unnamed instances are supported, although creating bnodes is usually only sensible in the context of identifying associations. For example, suppose that Abraham owned an axe and we wished to capture that information. We might say, "Abraham **owns** (an **Axe**)." Similarly, if we did not know or did not wish to explicitly name the woman whom Abraham married we could have dropped "Mary" from the parenthetical phrase after "married". In either case a typed bnode would be created, identifiable only (without "insider information" about system-generated identifiers) by its association with the Abraham node.

3.2. *Patterns and variables*

Natural language questions often contain graph patterns. For example, one might ask, regarding the information in Fig. 1, "To whom is Abraham married?" "Robert is a child of whom?" If we were creating a graph query language from scratch, we might translate these question into queries such as "Abraham married ?" and "? child Robert" where the "?" indicates values to be retrieved from the graph in answering the query. In some graph query languages each placeholder is given a name and is called a variable. It is necessary to name the variables so that the graph patterns will be unambiguous. For example, suppose that we asked the question, "What is the birthdate of the person to whom Abraham is married?" Without names this pattern would be "Abraham married ? and ? birthdate ?" and it is unclear that there are two different variable bindings implied. One can easily construct more complex graph patterns that make it even clearer that we need to be able to match the unbound variable in one graph pattern element (node → edge → node) with that in another. In this case we mean "Abraham married ?p and ?p birthdate ?x". Note that the naming of the variables can be entirely arbitrary; it only matters that the same names are used when linking elements together. Using SPARQL syntax, where conjunction of graph elements is indicated by a dot, the example questions are (prefix definitions omitted):

(1) Select ?x where {<lincolns:Abraham><gen:married> ?x}
(2) Select ?p where {?p <gen:child><lincolns:Robert>}
(3) Select ?p ?x where {<lincolns:Abrahm><gen:married> ?p · ?p <gen:birthdate> ?x}

The "whom" in the English questions are pronouns that play the role of unbound variables; unbound because before our brain or the query engine processes the query no values are bound to the variables. Only in the results are the variables bound to lists of possible values for each variable. In these examples, the results from Fig. 3 would be:

(1) x is Mary (a table with one column labeled "x" and one row with value Mary)
(2) p is {Mary, Abraham} (a table with one column labeled "p" and two rows, Mary and Abraham)
(3) an empty table with columns "p" and "x"; empty because Mary has no birthdate in the data graph of Fig. 3

In SADL graph patterns can be expressed using simple names (that are not already used for concepts) as variables. Names interpreted as variables are colored pink. If the values to be returned are unambiguous, the "select . . . where" can be dropped. A graph pattern to be treated as a query is preceded by "Ask:". Hence the queries above expressed in SADL are:

(1) Ask: Abraham **married** x.

(2) Ask: **P child** Robert.
(3) Ask: select **p**, **x** where Abraham **married p** and **p** has **birthdate x**.

The select variables are explicit in the 3rd example because we want both variables to be represented in the table of returned values.

Thinking of graph patterns in terms of named variables is not very much like natural language. One of the ways that English avoids using variables while avoiding ambiguity is through a chaining of triple patterns. For example, we might say, "What is the birthdate of the child of Abraham?" Note that the only pronoun in this sentence is the interrogative pronoun "what", which acts as a variable in this question. To better understand chaining of triple patterns, consider three ways of stating a triple in SADL, using "s p o" as the triple where "s" is the subject, "p" is the predicate, and "o" is the object.

(1) *s* [has | with] *p o* (ex: Abraham has child Robert) (the "has" or "with" are optional)
(2) [a | the] *p* of *s* is *o* (ex: a child of Abraham is Robert) (the "a" or "the" is optional)
(3) *o* is [a | the] *p* of *s* (ex: Robert is a child of Abraham) (the "a" or "the" is optional)

Note that alternate forms (2) and (3) do not work well with all predicates, e.g. the predicate "married" does not make a natural sounding phrase with syntax (2) and (3).

Revisiting the question "What is the birthdate of the child of Abraham?", the SPARQL equivalent is "select ?bd where {<lincolns:Abraham><gen:child> ?c . ?c <gen:birthdate> ?bd}". Taking these two triple patterns and rewriting them in the form 3) above:

(1) ?c is the child of Abraham
(2) ?bd is the birthdate of ?c

The first of these two statements is like an equation: "?c = the child of Abraham". Substituting the right hand side of this equality in for "?c" in the second statement yields "?bd is the birthdate of the child of Abraham". Substituting the interrogative pronoun "what" for "?bd" gives us the original English question. We have successfully eliminated the variable "?c".

As a somewhat extreme example of chaining of triple patterns to eliminate any explicit variables, consider the graph pattern shown in Fig. 6. In English we might say "George was born at a location with the same latitude as Philadelphia, was born

a **Birth** (with **location** with **latitude** == **latitude** of Philadelphia) with **child** George (with **mother** == **mother** of a **Birth** with **child** Samuel) with **weight** == **weight** of a **Birth** with **child** Ruby

Fig. 6. A complex SADL graph pattern with all variables eliminated.

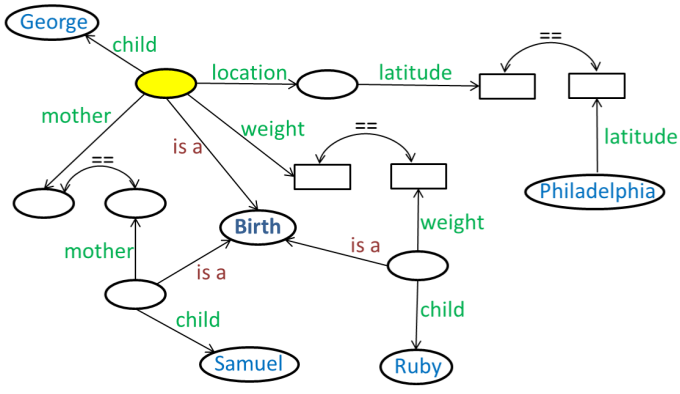

Fig. 7. A complex graph pattern with unbound variable nodes and equivalences.

to the same mother as Samuel, and had the same weight as Ruby." Figure 7 shows the pattern defined by the expression in Fig. 6 as a visual graph. In terms of a SPARQL query, the blank rectangles in Fig. 7 are unbound variables whose values will be literals, the blank ovals are unbound variables whose values will be instances, and the yellow oval is the default bindings returned if this expression is interpreted as a query. The equivalent symbols ("==") in the expression and graph translates into owl:sameAs for object properties (values are instances) or to a SPARQL equals filter for data properties (values are literals). The equivalent SPARQL query is (most namespaces eliminated)

```
Select ?b1 where {
?b1 <rdf:type><Birth> · ?b1 <location> ?loc1 · ?loc1 <latitude> ?l1 ·
<Philadelphia><latitude> ?l2 · ?b1 <child><George> · ?b1 <mother> ?m1 ·
?b2 <rdf:type><Birth> · ?b2 <mother> ?m2 · ?m1 <owl:sameAs> ?m2 ·
?b2 <child><Samuel> · ?b1 <weight> ?w1 · ?b3 <rdf:type><Birth> ·
?b3 <weight> ?w2 · ?b3 <child><Ruby> ·
FILTER (equals(?l1, ?l2) && equals(?w1, ?w2))}
```

Graph patterns and variables are also used in expressing rules using the concepts defined in the domain model. SADL has a plug-in architecture allowing any compatible reasoner, rule engine, and query engine to be used as long as a translator from SADL to the required representations is also provided. The default reasoner/rule engine/query engine for the SADL-IDE is OWL/Jena/ARQ: OWL for the domain model, Jena Rules [2] for rules, and SPARQL for queries. Alternatives would be Pellet, using OWL/SWRL/SPARQL, and AllegroGraph [1] using RDF/Prolog/(Prolog or SPARQL).

To illustrate a rule using the t-box model of Figs. 4 and 5, we add one additional property definition for "uncle" and then create a rule that infers uncle relationships from class membership (Man) and "child" relationships. The definition and the rule in SADL are shown in Fig. 8. The rule element "p != s" is necessary so that a father

```
uncle describes Person with values of type Man.

Rule UncleRule:
        given    s is a Man
        if       gp has child s
                 gp has child p
                 p != s
                 p has child c
        then     c has uncle s.
```

Fig. 8. Definition of "uncle" property and rule to infer containing statements.

is not inferred to be an uncle of his child. The colon after the rule name is optional as is the formatting on one or multiple lines. The "given" triple patterns normally translate to the rule premises just like "if" patterns and is syntactic sugar. For comparison, the Jena rule created by the translator from this SADL rule is shown in Fig. 9. The graph patterns used in rules can also be made more human-readable by eliminating variables through chaining of triple patterns. The genealogy example of Fig. 5 does not lend itself to an example so we will draw one from the domain of geometric shapes. Figure 10 shows a rule, in SADL and Jena format, for the area of a parallelepiped.

```
[UncleRule: (?s rdf:type http://sadl.org/TestSadlIde/genealogy#Man),
(?gp http://sadl.org/TestSadlIde/genealogy#child ?s),
(?gp http://sadl.org/TestSadlIde/genealogy#child ?p), notEqual(?p, ?s),
(?p http://sadl.org/TestSadlIde/genealogy#child ?c)
-> (?c http://sadl.org/TestSadlIde/genealogy#uncle ?s)]
```

Fig. 9. Jena rule equivalent of the SADL rule in Fig. 8.

```
Rule AreaOfParallelepiped:
        if x is a Parallelepiped
        then the volume of x is the height of x* the width of x* the depth of x.

[AreaOfParallelepiped:
(?x rdf:type http://sadl.org/TestSadlIde/temp#Parallelepiped),
(?x http://sadl.org/TestSadlIde/temp#height ?v0),
(?x http://sadl.org/TestSadlIde/temp#width ?v1), product(?v0, ?v1, ?v2),
(?x http://sadl.org/TestSadlIde/temp#depth ?v3), product(?v2, ?v3, ?v4)
-> (?x http://sadl.org/TestSadlIde/temp#volume ?v4)]
```

Fig. 10. Rule, SADL and Jena formats, to compute the area of a parallelepiped.

The kind of chaining here is a little different than in the queries above. Form (3) of the triple pattern, e.g. "? is height of x", is taken as an equivalence and "height of x" is substituted into the multiplication operation. The result is the same — the only variable in the SADL rule is that which is bound to instances of the class parallelepiped. In the translation shown in the bottom of Fig. 10, all implied variables, of which there are four, are made explicit.

4. Modeling Affordances Provided by the SADL-IDE

Regardless of the modeling representation, effective tools are an important resource for model developers and for overall life-cycle support. The SADL-IDE encompasses a number of functionalities that support model development and maintenance.

4.1. *Valid model is immediately computable, queryable and testable*

One of the useful features of the SADL-IDE, although it is not unique to SADL, is that once a valid model is built it is immediately a computable artifact that can be reasoned over by a reasoner/rule engine, queried by a query engine, and tested in the IDE. By tested we mean that we can express an expected conclusion and have the IDE verify that that conclusion is in fact reached. This immediacy is useful in rapid model development. To illustrate this point, suppose that we have the domain model defined in Fig. 5 with the content of Fig. 8 added. We will add the additional information that Thomas is Abraham's father and that Tommy is Abraham's brother. The complete instance data model, including the snippet previously shown in Fig. 3, is shown in Fig. 11. Note that the SADL file includes several queries and two tests. The queries simply return data asserted in the model. The two tests check to make sure that expected inferences have occurred. The first is true because of an OWL entailment: every instance of a Man (or a US_President) is also an instance of a Person. The second is concluded by the rule previously discussed.

Once this data graph has been entered into the SADL-IDE, it can be "executed" with a single keystroke (alt-shift-t). The results of such an execution are shown in Fig. 12. Through preference settings, various additional information including derivations (discussed below) and reasoner timing information can be displayed as well.

4.2. *Continuous error checking and contextual help*

Marking any errors or warnings found while processing the model is very helpful to modelers. For example, suppose that we define a property with domain Person and range Gender, but Gender has not been defined. The error we get in the SADL-IDE is shown in Fig. 13 (caused by commenting out the definition of Gender on line 10).

Note that two quick fixes are available. The first will define Gender as a top-level class, inserting the new defining statement into the model. The second will define

```
uri "http://sadl.org/TestSadlIde/Lincolns"
alias lincolns version"$Revision:$ Last modified on $Date:$".

import "http://sadl.org/TestSadlIde/genealogy".

Abraham is a {Man and US_President},
married (a Woman Mary with child
                                (a Person Robert
                                with birthdate "August 1, 1843")),
has child Robert.

Thomas is a Person with child Abraham, with child (a Man Tommy).

Ask: Abraham married x.
Ask: P child Robert.
Ask: a child of Mary.
Ask: select p, x where Abraham married p and p has birthdate x.
Ask: select d where d is birthdate of child of Abraham.

Test: Abraham is a Person.
Test: Robert has uncle Tommy.
```

Fig. 11. Extended instance data in SADL with queries and tests.

```
Results for query 'select x where Abraham, married, x':
    x = Mary

Results for query 'select P where P, child, Robert':
    P
    Abraham
    Mary

Results for query 'select v0 where Mary, child, v0':
    v0 = Robert

Results for query 'select p x where (Abraham, married, p) and (p, birthdate, x)':
  no results found

Results for query 'select d where (Abraham, child, v0) and (v0, birthdate, d)':
    d = 1843-08-01

Test passed: Abraham, rdf:type, Person

Test passed: Robert, uncle, Tommy

Passed 2 of 2 tests.

Ran 5 queries.
```

Fig. 12. Results of "executing" the model in Fig. 11.

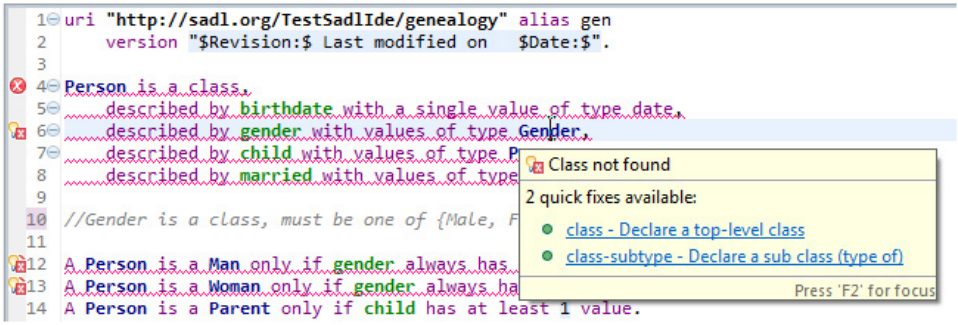

Fig. 13. Error markers and two quick fixes for missing definition of Gender.

Gender as a sub-class of something. If that quick fix is chosen, the inserted statement will highlight the name of the super-class, allowing the modeler to simply type the name or request a list of possible super-classes.

A good IDE will not only identify possible mistakes and bring them to the author's attention, the equivalent of continuous spell checking in a word processor, but will assist the user by substituting recognition for recall. A good example is "code completion" in a procedural language. Either automatically when input is paused, or on demand, the IDE will provide a list of possible completions of the current partial statement or templates of possible statements. For example, if the name of an instance of a class is typed followed by a dot, a Java IDE will provide a list of fields that can be accessed and methods that can be called. The modeler need only indicate a choice and the statement will be completed. Suppose that in the SADL-IDE we were specifying the test of the next to the last line in Fig. 11 and we were unsure of the class name. By pressing cntrl-space with the cursor at the end of the line (line 18 in Fig. 14), a list of available class names appears and the appropriate one can be

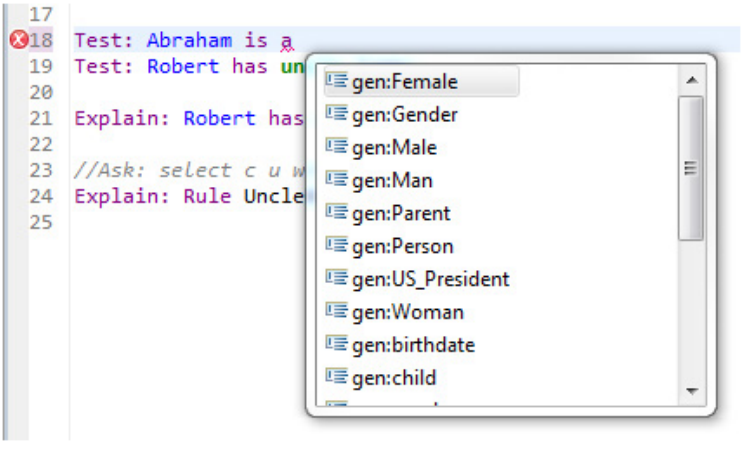

Fig. 14. Result of content assistance request in incomplete test statement.

selected, as shown in Fig. 14. The display of prefixes is controlled by a preference setting.

4.3. *Navigation*

Another example of modeler support is the hyperlinking of a name to its definition. This allows an author to easily navigate from any place where a concept is used to its definition. Or perhaps the definition is shown in a popup window when the cursor is placed over the name. In the SADL-IDE, a right mouse click and selection with the cursor over a concept other than in its definition will cause the model containing the definition to be opened, if the definition is not in the current model, and the definition will be brought into view. These and many other authoring aids make modelers more productive and reduce the number of errors in their models. The SADL-IDE contains a number of useful supporting functions and more are planned for the future.

4.4. *Versioning*

One of the important things we have learned from decades of software development, if not from even earlier experiences with documents such as legal contracts, is that it is important to clearly identify version information so as to be able to distinguish between subsequent versions of an artifact, especially one that is easily and often rapidly changing. Sophisticated version control systems now exist and are integrated into IDEs such as Eclipse. Version control systems that will be familiar to programmers include Concurrent Versions System (CVS), Subversion (SVN), and Git. These systems embed version information within code files or documents, allow differences between versions to be easily identified, manage the merging of modifications made in parallel by multiple modelers, allow branching and subsequent merging of the version tree, support the tagging of a set of code files or documents as a named set, etc. This kind of support is essential to successfully managing the evolution of a model or set of models over their useful life. The SADL-IDE offers all of these capabilities by virtue of being Eclipse-based.

4.5. *Semantic checking and regression testing*

In logical models, and increasingly in software models as well, tools often check the models not just for syntactic validity (the equivalent of spelling and grammar checking) but for semantic validity as well. For example, validation of an ontology might include checking to see if any class is defined and restricted in such a way that no instance could ever belong to the class. Ideally the tools would tell us if the model makes sense. The SADL-IDE provides some level of semantic checking, although much more is needed. Figure 15 shows the warning message when the same property is restricted to a single value of a given type in multiple statements. This alerts the user that qualified cardinality constraints are not supported as SADL currently uses OWL 1.0.

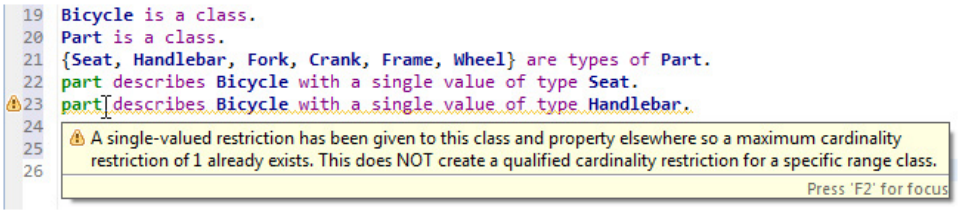

Fig. 15. Warning regarding a possible semantic error in the model.

Since such capability is difficult to achieve, less capable but still very useful approaches are often used. One is to develop a suite of comprehensive tests that exercise a model and determine that for a given set of inputs the model generates the correct outputs. A model that includes tests is shown in Fig. 11. A test suite is a simple file that identifies all of the models which should be tested, meaning their tests should be executed. When a test suite is executed, a summary of tests failing and tests passing is reported. In addition, the output of the test execution includes hyperlinks to each test executed. Such a test suite allows the model to be exercised regularly and/or whenever any change is made to the model in order to make sure that the model still produces the correct results for a variety of inputs. It is not the purpose of this paper to go into the details of test generation and coverage, but we do note that continuous testing is a proven way of increasing the quality and reliability of computational models.

4.6. *Debugging derivations*

A companion capability that is essential in a modeling tool suite is the ability to figure out why a test is failing. Debugging capability allows the modeler to analyze the computation occurring in a given test case scenario and understand where model behavior deviates from that expected by the test case. For procedural code this is often done by allowing the modeler to set break points in the code and step through the code in various ways, examining the values of variables at each step. In declarative models it makes less sense to analyze the process, as this is the process of a general purpose reasoner, and more sense to answer questions such as "How was this statement inferred?" or "Why wasn't this statement inferred?" While it is very important to be able to analyze why an incorrect answer was inferred by a logic-based model, it can be even more difficult to analyze why a desired statement was not inferred. An answer to either of these questions can be obtained by the following SADL statement.

Explain: Robert has **uncle** Tommy.

If the statement is true and was inferred, the derivation of the statement (the chain of rules and matching conditions that resulted in the inference) will be displayed. If the statement is not true, a search will be made for rules which could infer

this statement and each of those rules will be checked to see if their premises are matched in the current data graph. (Note that this check does not currently handle rule built-in functions.)

A more precise diagnostic statement for use when a rule is expected to fire but does not is the following SADL statement.

Explain: Rule UncleRule.

This statement will cause a check to be made to see how the premises of the rule are or are not satisfied in the current data graph.

5. Other Controlled-English Languages

Over the years, a large number of Controlled Natural Languages have been designed to address varying knowledge representation and communication needs. A very through survey of 99 English-based Controlled Natural Languages can be found in [10]. This work also provides a framework for classifying and comparing these languages. This classification is done on 4 dimensions: precision, expressiveness, naturalness and simplicity and is abbreviated as PENS. Each dimension in turn is assigned a value from one to five. To illustrate this classification scheme, English is classified as $P^1E^5N^5S^1$ and propositional logic is classified as $P^5E^1N^1S^5$; representing two extreme points.

One interesting English-based Controlled Natural Language is the Attempto Controlled English (ACE) [5]. It is a subset of English defined by associated syntax and interpretation rules. Its intended use is similar to our motivation for SADL; allowing people to specify formal models without requiring programming skills. ACE text is translated into Discourse Representation Structures (DRS) which are a syntactic variant of first-order logic [6]. Each noun in an ACE text must be introduced by a determiner, and this determiner allows mapping to a universal or an existential quantifier. Interpretation rules are used for entity reference resolution. ACE has been a part of the Attempto project at the University of Zurich since 1995 and version 6.6 was released in 2011.

ACE is classified as $P^4E^3N^4S^3$ [10] and we believe that SADL is $P^5E^3N^4S^4$. A value of 4 for precision corresponds to languages which are or can be defined by a formal grammar but whose representations may require background axioms or heuristics, etc., for deductions. A value of 5 for precision is assigned when the language is fully specified both at the syntactic and semantic level and each text has exactly one meaning that can be automatically derived. In regard to simplicity, a value of 3 corresponds to languages with lengthy descriptions — the language can be defined but requires more than 10 pages. A simplicity value of 4 corresponds to a language with a short description; more than one page and less than 10 pages. (Please note that we have done more in these pages than describe the SADL language.) The ANTLR grammar file for SADL is about 620 lines, including comments and blank lines and many short, indented lines for readability.

6. Some Examples of How SADL Has Been Applied

We have been using SADL for a number of years and it has also been made available as Open Source [18]. In this section we describe briefly and provide references to papers describing three SADL applications that illustrate and highlight different needs that were addressed.

6.1. *Modeling multi-level secure domains*

The first application involved modeling multi-level secure domains and how data provenance can flow and be manipulated as it crosses security boundaries [11–13]. This work required the development of complex models to capture the domain knowledge and even more complex rules to capture and manipulate confidentiality, integrity and assurance properties around data. We used the developed framework to model the proto-typical sensor-analyst-shooter scenario where the information flows from higher security level domains to lower security level domains as illustrated in Fig. 16 [12].

6.2. *Semantic model of smart grid*

The second application applied semantic models to the smart grid domain by providing rules for network traceability. This application showed how we could extend and build upon existing models (the CIM network model of NIST) [16] and how we

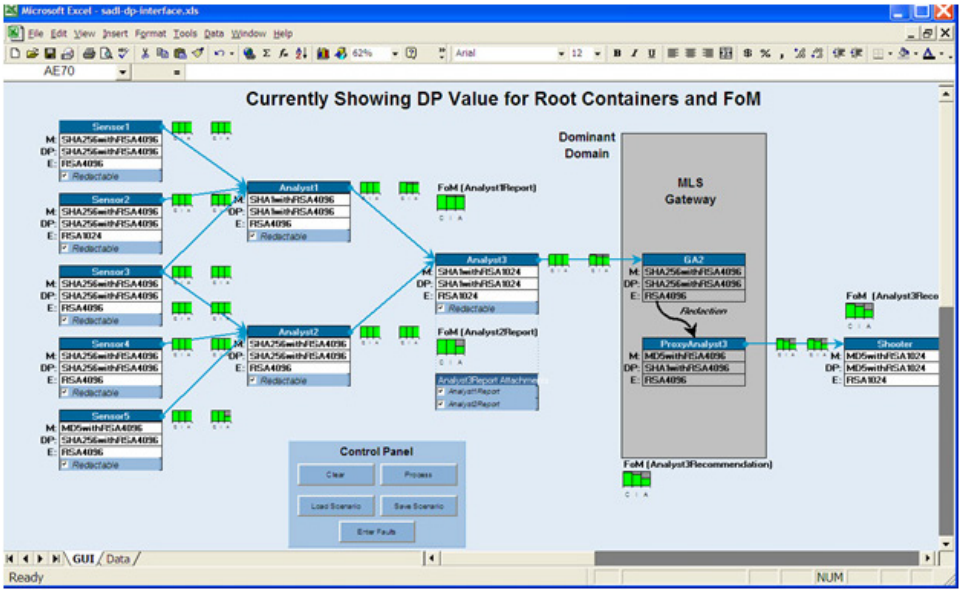

Fig. 16. Data Provenance (DP) and its Figure of Merit (FoM) in multi-level secure domains.

```
uri "http://sadl.imp/GridInteropExample".
 import "SelectedCim.sadl" as SelectedCim.

 // Desired relationship of a Breaker to a Disconnector on each side:
 //     Disconnector1 --toConnect--> TerminalD1C1
 //    --connectivityNode--> ConnectivityNode1
 //    --terminal--> TerminalC1B --fromConnect--> TheBreaker
 //    --toConnect--> TerminalBC2 --connectivityNode --> ConnectivityNode2
 //    --terminal--> TerminalC2D2 --fromConnect--> Disconnector2

 Rule Breaker IsolationConforms
  given b is a Breaker
  if    e1 is toConnect of connectivityNode of terminal of fromConnect of b
        e2 is fromConnect of terminal of connectivityNode of toConnect of b
        e1 is a Disconnector
        e2 is a Disconnector
  then
        isolationCompliance of b is true.

 Rule BreakerIsolationFromViolation
  given b is a Breaker
  if    e is toConnect of connectivityNode of terminal of fromConnect of b
        e is not a Disconnector
  then
        isolationCompliance of b is false.

 Rule BreakerIsolationToViolation
  given b is a Breaker
  if    e is fromConnect of terminal of connectivityNode of toConnect of b
        e is not a Disconnector
  then
        isolationCompliance of b is false.
```

Fig. 17. Importing and extending existing models.

were able to use AllegroGraph to do backward chaining for network traceability [3, 4]. Figure 17 shows a model that imports the CIM model and extends it with several rules that check for well-formedness of the network.

6.3. *Integrating semantic models with other tools*

The third application involved the integration of a semantic model with a CAD platform like Unigraphics so that we could bring together manufacturability rules and design rules in order to support design for manufacturability and reduce product design cost and lead-time [17]. An illustration of this is shown in Fig. 18 where rules corresponding to checking the design of sheet metal parts are integrated within a CAD system.

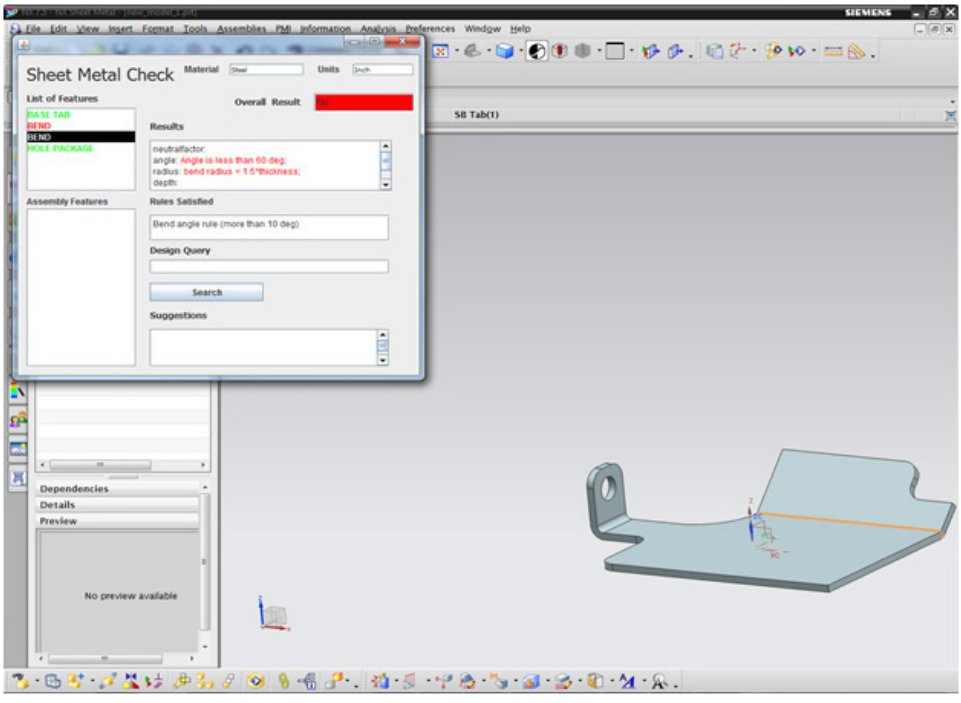

Fig. 18. Integrating rule checking within a CAD design tool.

7. Summary and Conclusions

The idea for SADL grew out of conversations at the Automated Software Engineering Conference in Atlanta in 2007. The syntax and grammar evolved as we built snippets of engineering models and showed them to SMEs. If the expert immediately started talking to us about what was right or wrong with the model and what else it needed to include, we considered that representation a success. If the expert asked us what the representation meant it was a failure. In other words, we wanted the domain experts to be able to at least understand a model, if not develop one, without having to think about the language. We were surprised by the degree to which we succeeded, at least at the understanding level. As we have described in this paper, the IDE support attempts to address the ease of authoring.

There are many more capabilities that could easily be added to the SADL language and environment. For example, it would be nice to be able to view the neighborhood of a class or instance as a visual graph. It is less clear that large, detailed graphs are really useful. A visual graph of the import hierarchy of a model with many constituents would also be very useful. These and many other support capabilities are very doable. It is only a question of resources. If SADL continues to find successful application, helping domain experts to have greater impact through

knowledge capture and knowledge services, we expect to continue our work. That is certainly our aspiration.

Acknowledgments

The authors would like to thank Michael Graham for his continuing support of this work.

References

[1] AllegroGraph. http://www.franz.com/agraph/allegrograph/.
[2] Apache Jena. https://jena.apache.org/.
[3] A. W. Crapo, X. Wang, J. Lizzi and R. Larson, The semantically enabled smart grid in *Grid Interop*, 2009, http://www.gridwiseac.org/pdfs/forum_papers09/crapo.pdf (accessed September 9, 2010).
[4] A. Crapo, K. Griffith, A. Khandelwal, J. Lizzi, A. Moitra and X. Wang, Overcoming challenges using the CIM as a semantic model for energy applications, in *Grid Interop*, 2010. http://www.smartgridnews.com/artman/uploads/1/crapo_gi10.pdf.
[5] N. E. Fuchs, K. Kaljurand and G. Schneider, Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces, in *FLAIRS* 2006.
[6] N. E. Fuchs, K. Kaljurand and T. Kuhn, Discourse representation structures for ACE 6.6, Technical Report 2010.0010, Department of Informatics, University of Zurich.
[7] P. N. Johnson-Laird, *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness* (Harvard University Press, Cambridge, MA, 1983).
[8] P. N. Johnson-Laird, *The Computer and the Mind* (Harvard University Press, Cambridge, MA, 1988).
[9] P. N. Johnson-Laird, *Human and Machine Thinking* (Lawrence Erlbaum Associates, Hillsdale, NJ, 1994).
[10] T. Kuhn, A survey and classification of controlled natural languages, Computational Linguistics, 2014, MIT Press.
[11] A. Moitra, B. Barnett, A. W. Crapo and S. J. Dill, Data provenance architecture to support assurance in a multi-level secure environment, in *MILCOM 2009*, 2009.
[12] A. Moitra, B. Barnett, A. W. Crapo and S. J. Dill, Addressing uncertainty and conflicts in cross-domain data provenance in *MILCOM 2010*, 2010, pp. 1764–1769. http://202.194.20.8/proc/MILCOM2010/papers/p1764-moitra.pdf.
[13] A. Moitra, B. Barnett, A. W. Crapo and S. J. Dill, Using data provenance to measure information assurance attributes, in *Provenance and Annotation of Data*, Revised Selected Papers/Vol. 6378/International Provenance and Annotation Workshop, Troy, NY, June 2010.
[14] Notation3 (N3): A readable RDF syntax. http://www.w3.org/TeamSubmission/n3/.
[15] OWL Web Ontology Language Reference: W3C Recommendation, 10 February 2004, available on-line at http://www.w3.org/TR/owl-ref/.
[16] S. Quirolgico, P. Assis, A. Westerinen, M. Baskey and E. Stokes, Towards a formal common information model ontology, *Web Information Systems — WISE 2004 Workshop*, LNCS, Vol. 3307, 2004, pp. 11–21.
[17] A. Rangarajan, P. Radhakrishnan, A. Moitra, A. W. Crapo and D. Robinson, Manufacturability analysis and design feedback system developed using semantic framework, in *Proc. ASME 2013 International Design Engineering Technical Conferences (IDETC)*

and Computers and Information in Engineering Conference (CIE), Aug 4–7, 2013, Portland, Oregon, USA.

[18] Semantic Application Design Language (SADL), Open Source project on Source Forge, overview at http://sadl.sourceforge.net/sadl.html.

[19] J. F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations* (Brooks Cole Publishing Co., Pacific Grove, CA, 2000).

[20] SPARQL Query Language for RDF: W3C Recommendation 15 January 2008, available online at http://www.w3.org/TR/rdf-sparql-query/.