The role of ontologies in creating & maintaining corporate knowledge: a case study from the aero industry

Derek Sleeman[1], Suraj Ajit[1], David W. Fowler[1], & David Knott[2]

[1]Department of Computing Science, University of Aberdeen, Scotland, UK

Email: d.sleeman@abdn.ac.uk, surajajit@yahoo.com, davidfowler0@googlemail.com

[2]Rolls Royce plc, Derby, UK

Email: david.knott@rolls-royce.com

Abstract. The Designers' Workbench is a system, developed to support designers in large organizations, such as Rolls-Royce, to ensure that the design is consistent with the specification for the particular design as well as with the company's design rule book(s). The evolving design is described against a jet engine ontology. Design rules are expressed as constraints over the domain ontology. To capture the constraint information, a domain expert (design engineer) has to work with a knowledge engineer to identify the constraints, and it is then the task of the knowledge engineer to encode these into the Workbench's knowledge base. This is an error prone and time consuming task. It is highly desirable to relieve the knowledge engineer of this task, and so we have developed a tool, ConEditor+ that enables domain experts themselves to capture and maintain these constraints. The tool allows the user to combine selected entities from the domain ontology with keywords and operators of a constraint language to form a constraint expression. In order to appropriately apply, maintain and reuse constraints, we believe that it is important to understand the assumptions and context in which each constraint is applicable; we refer to these as "application conditions". We hypothesise that an explicit representation of constraints together with the corresponding application conditions and the appropriate domain ontology could be used by a system to support the maintenance of constraints. In this paper, we focus on the important role the domain ontology plays in supporting the maintenance of constraints.

1 Introduction

The context for the principal system reported here, ConEditor+ (Ajit, Sleeman, Fowler, Knott, & Hui, 2005; Ajit, Sleeman, Fowler, Knott, & Hui, 2007), is the Designers' Workbench (Fowler, Sleeman, Wills, Lyon, & Knott, 2004) that has been developed to enable a group of designers to produce cooperatively a component that conforms to the component's overall specifications and the company's design rule book(s). One can view the design rule book(s) as an important repository of corporate knowledge, in a company whose expertise is principally in the design and maintenance of aero-engines. Moreover, we argue that the Designers' Workbench is an interactive environment in which this corporate knowledge is applied; further, ConEditor+ allows engineers to capture and maintain (verify and refine) these constraints. Further, as we shall demonstrate, an ontology for describing jet engines has a central role in these systems.

Engineering Design is constraint-oriented and much of the design process involves the recognition, formulation and satisfaction of constraints (Gross, Ervin, Anderson, & Fleisher, 1987; Lin & Chen, 2002; Serrano & Gossard, 1992; Ullman, 2003). The engineering design process has an evolutionary and iterative nature as designed artifacts often develop through a series of changes before a final solution is achieved. A common problem encountered during the design process is that of knowledge (e.g. constraint) evolution, which may involve the identification of new constraints or the modification or deletion of existing constraints. The reasons for such changes include development in the technology, changes to improve performance, changes to reduce development time and costs. Typically, maintenance involves various issues/problems:

- Original experts are unlikely to be available: The transient nature of modern organizations and workforces, the rapid flow of knowledge and experience out of companies due to staff leaving make it difficult for new designers to properly use stored design knowledge and subsequently to maintain it.
- Insufficient documentation provided: Several constraints may be applicable only in particular contexts. These contexts are often implicit to the designer formulating them but are not documented. In addition, many constraints are based on assumptions that have become false subsequently. These assumptions are often not made explicit.
- Maintenance is time consuming and complex: Maintenance of constraints in an engineering design environment is a complicated process that can be complicated and time consuming to perform manually. Thus, there is a pressing need for tools to support maintenance of this kind of knowledge.
- The evolutionary nature of constraints establishes the need to constantly update, revise, and maintain them. One needs to identify all the constraints that require modification. In addition, one needs to make sure that the knowledge base is consistent after making any changes.

The issues faced in Knowledge Base (KB) maintenance within engineering were first raised by the XCON configuration system at Digital Equipment Corporation

(Barker & O'Connor, 1989; Soloway, Bachant, & Jensen, 1987). Initially it was assumed that knowledge-based systems could be maintained by simply adding new elements or replacing existing ones. However this "simplicity" proved to be illusory as indicated by the experience of R1/XCON (Coenen, 1992).

1.1 Research Aims and Hypothesis

Enabling domain experts to maintain knowledge in a knowledge-based system has long been an objective of the knowledge engineering community (Bultman, Kuipers, & Harmelen, 2000). This paper identifies a situation where it is highly desirable to eliminate the knowledge engineer from doing this laborious, error-prone and time-consuming task. The paper reports on a system ConEditor+ that we have developed to enable domain experts themselves to capture and maintain constraints. Further, we hypothesize that it is important to capture the context in which a constraint is applicable in a system interpretable format and that this information (referred to as application conditions) together with the constraints and the domain ontology can be used by a system to support the maintenance of constraints. For us, the maintenance of constraints includes reducing the number of inconsistencies and also detecting redundancy, subsumption and fusion between pairs of constraints. In particular, we aim to exploit inferencing inherent in the domain ontology to support the maintenance of constraints. The main research question we plan to address is:

Can an explicit representation of application conditions together with the constraints and the domain ontology help a system: a) reduce the number of inconsistencies and b) detect subsumption, redundancy, fusion and suggest appropriate refinements between pairs of constraints?

The rest of the paper is organized as follows: Section 2 provides an introduction to the Designers' Workbench, and the domain ontology used; Section 3 describes the problem(s) faced in developing the knowledge base for the Workbench and the need for ConEditor and ConEditor+. Section 4 gives a brief overview of ConEditor+. Section 5 then focuses on the maintenance aspects of constraints with a description of our approach. Section 6 describes how we extended the jet engine ontology and then used this in the refinement of constraints from that domain. Implementation of ConEditor+ is discussed in Section 7. Evaluation undertaken is described in Section 8 and is followed by discussion of related work in Section 9. Conclusions and plans for future work follow in Section 10.

2 Introduction to the Designers' Workbench

Designers in Rolls-Royce, as in many large organizations, work in teams. Thus it is important when a group of designers are working on aspects of a common project, that the subcomponent designed by one engineer is consistent with the overall specification, and with those designed by other members of the team. Additionally, all designs have to be consistent with the company's design rule book(s). Making sure that these various constraints are complied with is a complicated process, and so we have developed the Designers' Workbench, which seeks to support these activities.

The Designers' Workbench (Fig. 1) uses an ontology (Gruber, 1995) to describe elements in a configuration task. Design rules are expressed as constraints over the ontology. The system supports human designers by checking that their configurations satisfy both physical and organizational constraints. Configurations are composed of features, which can be geometric or non-geometric, physical or abstract. When a new design is input into the system an engineering drawing is provided as a graphical backcloth, and the various parts are annotated using the domain ontology. Fig.1 shows the result of such an annotation exercise; the relevant ontology displayed in the top right hand corner can be expanded to show sub-classes, properties, and relations. A graphical interface enables the designer to add new features, set property values, and perform constraint checks. If a constraint is violated, the affected features are highlighted and a report is generated.

INSERT FIGURE 1 HERE

The report gives the designer a short description of the constraint that is violated, the features affected by that violation, and a link to the source document. The designer can often resolve the violations by adjusting the property values of the affected features. On selecting a feature, the GUI displays a table of corresponding properties and their values. These property values can then be adjusted, and this often resolves the constraint violation(s). The ontology used by the Designers' Workbench was created using the Protégé editor (Noy, Fergerson, & Musen, 2000), and the class hierarchy is shown in Fig. 2. The ontology is written in the Web Ontology Language (OWL) (McGuinness & Harmelen, 2004), and has 42 classes and 45 properties (of which 22 are object properties and 23 are data-type properties). Most classes in the ontology correspond to features, and the properties correspond to parameters that can be set to instances of feature (data-type properties), or to connections to other features (object properties).

INSERT FIGURE 2 HERE

Fig. 3 shows the properties of a class (`DiametralRingSeal`) selected from the ontology. There are three datatype properties (`in_static_joint`, `name`, and `owner`) and six object properties (`has_ferrule`, `has_housing`, `has_coating`, `has_material`, `has_sealing_ring`, and `operating_temperature`) that link to other entities in the design. Furthermore, two of the properties (`has_ferrule` and `has_housing`) are only defined for the class `DiametralRingSeal`, whereas the others are defined for classes that are ancestors of the class, and are inherited (as shown by the brackets around the rectangular icons in the screenshot).

INSERT FIGURE 3 HERE

In the Designers' Workbench, the designer can select a feature class from the ontology and create an instance of that class. The values of the properties of a typical instance of the class DiametralRingSeal are shown in Fig. 4. In the Designers' Workbench, property values are set by either typing values into a text box (for datatype properties), or by selecting an instance from a drop down menu (for object properties); also values can be left uninstantiated. This enables the designer to fill in the values that are known, and to check constraints, in an incremental way.

INSERT FIGURE 4 HERE

Example constraints defined over the ontology include:

- The value of the maximum operating temperature of the material of each concrete feature must be greater than the prevailing environmental temperature;

- The length of the bolt in a bolted joint must exceed the sum of the thicknesses of the clamped parts, plus the height of the nut. For simplicity, issues such as tolerances of dimensions have been ignored although these can be dealt with, for example by defining a Measurement class (as subclass of AbstractFeature), with properties dimension and tolerance.

The first constraint above will apply to all concrete features that have a ‘has_material’ property and an ‘environmental_temperature’ property defined. The second constraint above is more complicated, and applies to all bolts, nuts, and clamped parts that are parts of bolted joints. Constraints are handled in a two stage process:

- Identify feature values that should be constrained;

- Formulate a tuple(s) of values for each set of feature values, and check that the constraint is satisfied by these values.

The constraint processing uses SPARQL Query Language (Prud'hommeaux & Seaborne, 2007) to find the constrained features and values. After using SPARQL to extract the constrained values, SICStus[1] Prolog is used to check that the constraints hold. The SPARQL query that locates features affected by the material temperature constraint is:

```
SELECT ?arg1,?arg2 WHERE
(?feature,<dwOnto:has_material>,?mat),
(?mat,<dwOnto:max_operating_temp>,?arg1),
(?feature,<dwOnto:operating_temp>,?optemp),
(?optemp,<dwOnto:temperature>,?arg2)
USING dwOnto FOR <namespace>
```

[2]Swedish Institute of Computer Science, version 3.10, Accessed online 29 May 2008 at http://www.sics.se/sicstus/

The values of the returned variables `?arg1` and `?arg2` are the material's maximum operating temperature, and the current operating temperature, respectively. The check that the values must satisfy is represented by the Sicstus predicate

```
op_temp_limit(MaterialMaxTemp, EnvironTemp) :-
EnvironTemp =< MaterialMaxTemp.
```

Using the values of the variables `?arg1` and `?arg2`, the predicate `op_temp_limit(MaterialMaxTemp, EnvironTemp)` is formed, and checked. This process is repeated for each set of values returned by the SPARQL query, and for each constraint that has been specified.

INSERT FIGURE 5 HERE

3 Capturing the knowledge in the design rule books

As noted above, the Designers' Workbench needs access to the various constraints, including those inherent in the company's design rule book(s). To capture this information, a design engineer (domain expert) worked with a knowledge engineer to identify the constraints, and it was then the task of the knowledge engineer to encode these into the Workbench's knowledge base. This was an error prone and time consuming task. As constraints are explained succinctly in the design rule book(s), a non-expert often finds it very difficult to understand the context and formulate constraints directly from the design rule book(s), and so a design engineer has to help the knowledge engineer in this process. An example of a constraint as expressed in the rule book(s) is shown in Fig. 5.

It would be useful if a new constraint could be formulated by an engineer in an intuitive way, by selecting classes and properties from the appropriate ontology, and somehow combining them using a predefined set of operators. This would help engineers to input the constraints themselves and relieve the programmer of that task. This would also enable designers to have greater control over the definition and refinement of constraints, and presumably, to have greater trust in the results of constraint checks. This led initially to the development of a system, known as ConEditor (Ajit, Sleeman, Fowler, & Knott, 2004), which enables a domain expert to input and maintain constraints. ConEditor concentrated on the constraint capture and provided basic maintenance facilities such as syntax error checking, allowing users to read constraints from a file, edit the constraints and then write them to the same or a new file. Following encouraging results from a preliminary evaluation undertaken at Rolls-Royce, ConEditor was enhanced with additional features to support the maintenance of constraints and became known as ConEditor+. The paper refers to the latest version of the system, ConEditor+, throughout the paper. Details of the system and its maintenance features are given in subsequent sections.

INSERT FIGURE 6 HERE

4 ConEditor+

ConEditor+ is a system that has been developed to enable domain experts to capture and maintain constraints. ConEditor+'s graphical user interface (GUI) is shown in Fig. 6. A constraint expression can be created by selecting entities from the taxonomy (domain ontology) and combining them with a pre-defined set of keywords and operators from the high level constraint language, CoLan (Bassiliades & Gray, 1995; Gray, Hui, & Preece, 2001). CoLan has features of both first-order logic and functional programming, and was designed to enable scientists and engineers to express constraints in a computer environment themselves.

ConEditor+'s GUI essentially consists of six components, namely: (A) Keywords Panel, (B) Menu Bar, (C) Functions Panel, (D) Taxonomy / Ontology Panel, (E) Tool Bar and (F) Result Panel (see Fig. 6). The user can then select the appropriate entities with the mouse and so form a constraint expression. The taxonomy in the top right hand window (displayed again separately in

Fig. 7) shows that the class under discussion is `ConcreteFeature`, and the class `ConcreteFeature` contains the various properties `has_coating`, `has_lubricant`, `has material`, etc. Each property has a range class which, in turn, consists of more properties (e.g., `has_material` is a property that has the range class `Material`; further the class `Material` has properties `density`, `max_operating_temp`, etc.). The Taxonomy/Ontology Panel is used to select entities from the domain ontology. More details about the GUI can be found in (Ajit, 2008). An analysis of the Rolls-Royce's design rule book(s) showed that a number of constraints are expressed in tables and so ConEditor+ provides a mechanism for inputting tables. When a constraint is modified and saved, ConEditor+ stores the modified constraint as a new version together with the original constraint. Storing all the versions would enable designers to study the evolution of constraints. Each constraint is allocated a unique identification number (ID) that includes its version number. The system provides facilities to retrieve constraints using keyword-based searches, e.g. search and retrieve all the constraints containing the specified keyword(s) or find the constraint with the specified ID.

INSERT FIGURE 7 HERE

5 Maintenance of constraints

Due to restricted availability of Rolls-Royce designers' time and for simplicity, we initially used a kite domain for our study (Eden, 1998; Streeter, 1980; Yolen, 1976) and so developed an ontology for kite design. In order to explain the concept of application condition, we consider the following constraint from the kite domain together with its associated application condition:

Constraint – "The density of the cover material of the kite must be greater than 0.5 ounces per square inch."

Application condition – "This is applicable only when there is a requirement to produce low cost kites for beginners. Kites for experts use lighter materials that are of higher quality and hence costlier."

As shown in the example above the application condition specifies the context in which the constraint is applicable. In order to tackle the various maintenance issues, our approach has the following stages:

- Capture the "context" of a constraint, in a machine interpretable form, as an application condition associated with the constraint.
- Use the application condition together with the constraint and the appropriate domain ontology to support the maintenance of constraints

We have extended ConEditor+ so that the user (the domain expert) can associate an application condition with each constraint. Often, such information is implicit to the person who formulates the constraint. We believe that it is important to make the application conditions explicit so that they can be used for both maintenance and reuse of constraints. The assumptions on which a constraint is based may no longer be true and in such cases, it becomes necessary to deactivate or remove those constraints from the KB. Further, an application condition may not be relevant to a particular design task.

ConEditor+ captures both the constraints and the application conditions in the same language, CoLan. Both are then converted into a standard machine interpretable format known as Constraint Interchange Format (CIF) (Gray *et al.*, 2001). We give below a typical constraint and its application condition in CoLan:

```
constrain each k in Kite
such that has_type(k) = "Flat" and has_shape(k) = "Diamond"
to have tail_length(has_tail(k)) = 7 * spine_length(has_spine(k))
```

In the above constraint, the application condition (in italics) is introduced by the clause "`such that`". This constraint states that the length of a tail of a kite needs to be seven times the length of the spine of the kite; however, this constraint is only applicable to flat diamond-shaped kites.

In order to make it clear, we divide a constraint in CoLan into three parts namely antecedent, application condition and consequent. Thus, the above constraint consists of:

Antecedent: `constrain each k in Kite`

Application condition: `such that has_type(k) = "Flat" and`

```
                  has_shape(k) = "Diamond"
```

Consequent: `tail_length(has_tail(k)) = 7 *`

```
        spine_length(has_spine(k))
```

The clause "`such that`" is a part of CoLan language and it is used to express conditional statements. We have currently made use of this clause to represent application conditions. As part of the future work, we plan to change the naming conventions used by the properties in the ontology and also extend the CoLan language to include "`when`" instead of "`such that`" for enhanced readability. The above constraint can then be expressed alternatively as follows:

```
constrain each k in Kite
to have tail_length_of(tail_of(k)) = 7 *
spine_length_of(spine_of(k))
when type_of(k) = "Flat" and shape_of(k) = "Diamond"
```

INSERT FIGURE 8 HERE

6 Extension of Jet Engine Ontology and Maintenance of a more Complex Set of Constraints

After a successful application and evaluation of ConEditor+ in the domain of kite design, we decided to apply our approach to part of the considerably more demanding Rolls-Royce domain. As the initial Rolls-Royce KB (used by the Designers' Workbench) only covered a small part of the engine, it was decided to review some additional design rule books, and interviews were held with an appropriate domain expert at Rolls-Royce. We then extended the jet engine ontology to incorporate the additional concepts and properties obtained from these analyses.

Fig. 8 shows a screenshot of the extended jet engine ontology developed using Protége. We then expressed all the constraints together with their application conditions against the extended jet engine ontology. There are a number of ways in which we can use the domain ontology together with the constraints and application conditions to support the maintenance of constraints. Refinement of the constraint KB is described, in some detail, below.

6.1 Redundancy

Redundancy occurs between constraints when all the components of a constraint (antecedent, application condition and consequent) are equivalent to the corresponding components of another constraint. Two types [(a) and (b)] of redundancy that make use of inferences from the domain ontology are described as follows:

(a) Using Class Equivalence

In OWL (Bechhofer *et al.,* 2004), `owl:equivalentProperty` is a built-in property that links a class description to another class description such that the two class descriptions involved have the same class extension (i.e., both class extensions contain exactly the same set of individuals). Consider constraints of the following form:

(i)
```
constrain each c in C1
such that X
to have Y
```

(ii)
```
constrain each c in C2
such that X
to have Y
```

In the constraints above, `c` is a variable, C_1 and C_2 are classes, `X` and `Y` are properties. If C_1 is an equivalent class (i.e. `owl:equivalentClass`) to C_2 in the domain ontology, one can infer that the constraint (i) is equivalent to constraint (ii). ConEditor+ notifies the user (domain expert) of this fact and suggests that the user considers eliminating this redundancy.

(b) Using Property Equivalence

In OWL (Bechhofer *et al.,* 2004), `owl:equivalentProperty` is a built-in property that is used to state that two properties have the same property extension (i.e. both properties contain exactly the same set of values). Consider constraints of the following form:

```
(iii)   constrain each c in C
        such that X1
        to have Y
```

```
(iv)    constrain each c in C
        such that X2
        to have Y
```

In the constraints above, `c` is a variable, `C` is a class, X_1, X_2 and `Y` are properties. If X_1 is an equivalent property (i.e. `owl:equivalentProperty`) to X_2 in the domain ontology, one can infer that the constraint (iii) is equivalent to constraint (iv). ConEditor+ notifies the user (domain expert) of this fact and suggests that the user considers eliminating this redundancy.

6.2 Subsumption

Subsumption occurs between a pair of constraints when one constraint "covers" all the conditions of another constraint i.e. constraint A subsumes constraint B, if B is satisfied whenever A is satisfied. A type of subsumption that makes use of inferences from the domain ontology is described as follows:

The `rdfs:subClassOf` construct is defined as part of Resource Description Framework (RDF) Schema (Brickley & Guha, 2004). It is used in OWL and the meaning is exactly the same, i.e., if the class description C_1 is defined as a subclass of class description C_2, then the set of individuals in the class extension of C_1 should be a subset of the set of individuals in the class extension of C_2. Consider constraints of the following form:

```
(v)     constrain each c in C1
        such that X
        to have Y

(vi)    constrain each c in C2
        such that X
        to have Y
```

In the constraints above, `c` is a variable, C_1 and C_2 are classes, `X` and `Y` are properties. If C_2 is a subclass (i.e. `rdfs:subClassOf`) of C_1 in the domain ontology, one can infer that the constraint (v) subsumes constraint (vi). ConEditor+ notifies the user (domain expert) of this fact and suggests that the user removes / deactivates constraint (vi).

6.3 Fusion of Classes

Fusion occurs between a pair of constraints when the two constraints can be combined together and replaced with another constraint, i.e. two constraints A and B can be fused together and replaced by a constraint C if C is satisfied in the same situations that A and B are both satisfied. A type of fusion that makes use of inferences from the domain ontology is described below. Consider constraints of the following form:

(vii)
```
constrain each c in C1
such that X
to have Y
```

(viii)
```
constrain each c in C2
such that X
to have Y
```

In the constraints above, `c` is a variable, C_1 and C_2 are classes, `X` and `Y` are properties. Let C_3 be another class in the same domain ontology. If C_1 and C_2 are the only two sub classes (i.e. `rdfs:subClassOf`) of C_3 in the domain ontology, and if every instance (or individual) of C_3 is an instance of either C_1 or C_2, then the constraints (vii) and (viii) can be fused together and replaced by the constraint (ix) as follows:

(ix)
```
constrain each c in C3
such that X
to have Y
```

ConEditor+ notifies the user (domain expert) of this fact and suggests that the user considers fusing constraints (vii) and (viii) into (ix).

The reader is encouraged to refer (Ajit, 2008) for several other types of redundancy, subsumption and fusion detected by ConEditor+. Also, all types of refinements implemented in ConEditor+ are expressed in a formal notation and logically proved. Inconsistency (or contradiction) detected by ConEditor+ is described below in Section 6.4.

6.4 Inconsistency/Contradiction

An inconsistency/contradiction occurs between a pair of constraints when the consequent of one constraint contradicts the consequent of another constraint while the antecedents and application conditions are equivalent i.e., constraint A contradicts constraint B or vice-versa if both constraints A and B are unsatisfiable. An example of this type of inconsistency follows:

```
(x)     constrain each c in Component
        such that name(component_coating(c)) = "silver"
        to have tensile_strength(component_material(c)) < 1390

(xi)    constrain each c in Component
        such that name(component_coating(c)) = "silver"
        to have tensile_strength(component_material(c)) > 1590
```

By comparing the two constraints above, one can infer that the constraint (x) contradicts constraint (xi). ConEditor+ notifies the user (domain expert) of this fact and suggests that the user takes appropriate action (modify/delete) to resolve the inconsistency.

6.5 Overview

ConEditor+ can also deal with the situation where a constraint needs to have multiple refinements applied before it is possible to determine whether another constraint is equivalent, subsumed, inconsistent or fusible. For example, consider the following two constraints:

```
(xii)  constrain each s in SledKite
       such that has_level(s) = "beginner" or
       has_wind_condition(s) = "strong"
       to have kite_line_strength(has_kite_line(s)) > 30

(xiii) constrain each c in ConventionalSledKite
       such that has_class(c) = "beginner"
       to have kite_line_strength(has_kite_line(c)) < 25
```

If `ConventionalSledKite` is a subclass (i.e. `rdfs:subClassOf`) of `SledKite` and `has_level` is an equivalent property (i.e. `owl:equivalentProperty`) to `has_class` in the domain ontology, then it can be concluded that the constraint (xii) contradicts (xiii). ConEditor+ notifies the user (domain expert) of this type of inconsistency and suggests that the user takes appropriate action (modify/delete) to resolve the inconsistency.

The concepts and properties used in the constraints here are taken from the domain ontology. Hence the units used for all the measurements are to be defined in the domain ontology, instead of explicitly specifying them in each constraint. As part of the future work, we plan to integrate the domain ontology with the engineering mathematics ontology developed by (Gruber & Olsen, 1994) to incorporate physical dimensions, units of measure, etc. and enhance the ability to ensure that there is consistency between the units inherent in the constraints.

7 Implementation

Both the Designers' Workbench and ConEditor+ are implemented in the Java programming language. The domain ontology in OWL (McGuinness & Harmelen, 2004) is developed using the Protégé ontology editor (Noy *et al.*, 2000) and accessed using Jena (HP, 2000). ConEditor+ converts the ontology in OWL into an equivalent P/FDM Daplex schema (Bassiliades & Gray, 1995) using a transformation program developed in Java. This conversion is currently required as we have used an existing constraint language (CoLan) that was developed for databases. The Daplex schema is used by a Daplex compiler within ConEditor+ to detect any syntactic errors among constraints. The constraints are initially expressed in CoLan and then converted into a standard semantic web[2] enabled XML-Constraint Interchange Format (CIF) (Gray *et al.*, 2001). ConEditor+ uses Jena to interpret the CIF representation of constraints and application conditions together with the OWL domain ontology to detect inconsistencies and refinements between pairs of constraints. The inferences made from the domain ontology play an important role in detecting inconsistencies and refinements.

ConEditor+ performs a static comparison of pairs of constraint expressions, i.e. ConEditor+ compares constraints at the syntactical level, rather than comparing the solution sets. So ConEditor+ is comparing pairs of constraints of the form e.g. P(x1, x2) & Q(x1,x3,a) and P(x1, x2) & Q(x1,x3,b). By looking at the values of the constants (a,

[2] The semantic web is an evolving extension of the world wide web in which web content can be expressed in a form that can be understood, interpreted and used by computers to find, share and integrate information more easily. ((Berners-Lee, Hendler, & Lassila, 2001))

b), the structure of the predicates (P, Q), and inferring the semantic relationships between the corresponding classes and properties in the constraints from the domain ontology, ConEditor+ determines whether there is an inconsistency, subsumption, redundancy or fusion. When a constraint is submitted to ConEditor+, it is compared with each constraint in the KB. Hence the time complexity is O(n). More details about the implementation and complexity of ConEditor+'s algorithm can be found in (Ajit, 2008).

8 Evaluation

An experiment conducted to address the main research question is described below. The aim of this experiment was to address the research question:

Could an explicit representation of application conditions together with the constraints and the domain ontology help a system: a) reduce the number of inconsistencies and b) detect subsumption, redundancy, fusion and suggesting appropriate refinements between pairs of constraints?

We studied the kite design domain and captured constraints together with the corresponding application conditions (rationales). We ran an experiment with ConEditor+ using: (I) KB_1 containing 15 constraints together with their application conditions, (II) KB_2 containing the same constraints without any application conditions. The reader is encouraged to refer to (Ajit, 2008) for the complete list of constraints and the corresponding application conditions that have been captured from the kite design domain.

Results: For KB_1, ConEditor+ detected 3 subsumptions, 0 inconsistencies, 3 redundancies and 2 cases of fusion between pairs of constraints. For KB_2, ConEditor+ detected 2 subsumptions, 5 inconsistencies, 3 redundancies and 4 cases of fusion between pairs of constraints. The investigator confirmed that some of the inconsistencies, etc reported for KB_2 were spurious, and concluded that the absence of application conditions have caused these to be reported by ConEditor+ (5 contradictions and a number of inappropriate refinements). This is explained further below with the help of some examples. Let us consider two KBs, namely, KB_A and KB_B containing the following constraints:

KB_A (with application conditions):

(i)
```
constrain each k in Kite
such that has_level(k) = "beginner"
to have density(has_material(has_cover(k))) < 0.5
```

(ii)
```
constrain each k in Kite
such that has_level(k) = "advanced"
to have density(has_material(has_cover(k))) > 1.0
```

KB_B (without application conditions):

(iii)
```
constrain each k in Kite
to have density(has_material(has_cover(k))) < 0.5
```

(iv)
```
constrain each k in Kite
to have density(has_material(has_cover(k))) > 1.0
```

As shown above, the KB_A contains two constraints [(i) and (ii)] with the corresponding application conditions. The KB_B contains the same pair of constraints [(iii) and (iv)] without the corresponding application conditions. For KB_A, ConEditor+ does not detect any inconsistency (or contradiction). For KB_B, ConEditor+ detects an inconsistency between the two constraints [(iii) and (iv)]. Hence, it can be concluded that the absence of application conditions can cause a number of inconsistencies among constraints. Also, this can cause ConEditor+ to suggest inappropriate refinements as shown below: For example, let us consider two KBs, namely, KB_C and KB_D containing the following constraints:

KB_C (with application conditions):

```
(v) constrain each k in Kite
such that has_level(k) = "beginner"
to have bridle_length(has_bridle(k)) > 3 * has_height(k)
and kite_line_strength(has_kite_line(k)) > 90

(vi) constrain each d in Delta_kite
such that has_wind_condition(d) = "strong"
to have bridle_length(has_bridle(d)) > 3 * has_height(d)
```

KB_D (without application conditions):

```
(vii) constrain each k in Kite
to have bridle_length(has_bridle(k)) > 3 * has_height(k)
and kite_line_strength(has_kite_line(k)) > 90
```

```
(viii) constrain each d in Delta_kite
       to have bridle_length(has_bridle(d)) > 3 * has_height(d)
```

Again, two KBs have been considered: KB_C and KB_D, with and without application conditions respectively. `Delta_kite` is a subclass of `Kite` in the domain ontology. Hence, for KB_D, ConEditor+ inappropriately suggests the user (domain expert) considers deleting/deactivating constraint (viii) because constraint (vii) subsumes constraint (viii).

One can infer from the results of the experiment described above that an explicit representation of the application conditions together with the constraint reduced the number of inconsistencies and prevented ConEditor+ from suggesting inappropriate refinements. The results have demonstrated that an explicit representation of application conditions together with the constraints and the domain ontology could help a system in i) reducing the number of inconsistencies and ii) detecting subsumption, redundancy, fusion and suggesting appropriate refinements between pairs of constraints.

We performed usability studies of ConEditor at Rolls-Royce and obtained encouraging feedback from the design engineers. We also conducted an experiment to determine the usability of ConEditor+ using five subjects that included post-graduate engineering students from our university. The subjects were asked to answer an usability questionnaire and use a 5-point rating scale (1 being poor and 5 being excellent). The average rating given by the subjects was 3.8. The reader is encouraged to refer to (Ajit, 2008) for more information regarding the usability studies. Further, we conducted an experiment to determine the time taken by ConEditor+ to process constraints (including application conditions) and detect inconsistencies, redundancy, subsumption and fusion. Four KBs containing 30, 60, 90 and 120 constraints (including application conditions) respectively were considered. For each KB, two types of tasks were performed for each refinement to determine the worst case and best case time. To determine the best-case time, the KB was organized such that ConEditor+'s comparison of the submitted constraint with the first constraint in the KB resulted in an inconsistency/ refinement. To determine the worst-case time, the KB was organized such that ConEditor+'s comparison of the submitted constraint with the last constraint in the KB resulted in an inconsistency/ refinement. The time was recorded programmatically for each task. The experiment was run in a computer with the following configuration: AMD Athlon 64-bit processor, clock frequency of 2.21 GHz, 960 MB of RAM, operating system: Windows XP, JDK (Java Development Kit) 1.4.2 and Jena 2.1.

INSERT FIGURE 9 HERE

The time taken by ConEditor+ to report a syntax error in the submitted constraint was recorded programmatically and it was equal to 500 milliseconds. It can be observed from Fig. 9 that the average worst-case time taken by ConEditor+ for refinements essentially increases linearly as the KB size increases while the average best-case time taken is almost a constant. ConEditor+ uses Jena to parse the domain ontology, constraints and application conditions in CIF. Currently a file system (text files) is used to store the constraints. The increase in average worst-case refinement time could become non-linear for larger KBs that involve manipulation of information which cannot all be held in main memory. The semantic web technologies such as Jena face scalability issues, and work is being carried out by the semantic web community to tackle them. For large KBs containing thousands of constraints, we plan to use 3-store

(Harris & Gibbins, 2003) which is a RDF bulk storage and query engine developed within the AKT[3] project to enable the efficient handling of large RDF KBs. Moreover, although the total number of design constraints formulated by Rolls-Royce is in the order of thousands, we expect that only a small subset (say in the order of hundreds) will be needed for any particular design. With this number of constraints, our earlier results, suggest that speed should not be an issue.

9 Related Work

One of the first attempts to manage constraints for automation of computation in engineering applications was the work of (Harary, 1962) and (Steward, 1962). Since then there has been considerable amount of work done on the representation, use and management of constraints including the development of rule-based systems (Frayman & Mittal, 1987; Wielinga & Schreiber, 1997) and in the field of diagnosis (Felfernig, Friedrich, Jannach, & Stumptner, 2004). Constraint management done in systems above mainly refers to the detection of redundant and contradictory constraints during constraint solving whereas ConEditor+ detects redundant, subsumed, contradictory and fusible constraints *prior* to constraint solving. ConEditor+ compares pairs of constraints by looking at the values of the constants, and the structure of the predicates rather than by computing the solution sets of constraints. It became important to represent the defaults and preferences declaratively as constraints, rather than encoding them in the procedural parts of the program (Borning, Maher, Martindale, & Wilson, 1989). In most cases, domain-oriented or method-oriented tools (in the form of templates) were provided to capture constraints/rules from the domain experts. The cost of developing such tools is high, especially when their restricted scope is taken into account (Eriksson

[3] Advanced Knowledge Technologies Project. More information on http://www.aktors.org/.

et al., 1995). In comparison to the above tools, ConEditor+ is a domain independent tool that can be used by domain experts to capture constraints using the appropriate domain ontology. These constraints are converted into a standard format (in CIF) for use by other systems. A similar tool for capturing constraints has been developed by (Gray & Kemp, 2006) for database schemas. This tool uses a diagrammatic representation in the form of a relationship graph to capture constraints. The principal disadvantage of this tool is that the diagram can become cumbersome for large database schemas.

CoLan is similar to the constraint language Galileo (Bowen, O'Grady, & Smith, 1990) that has been used to support conceptual design and design knowledge representation. Both CoLan and Galileo are based on first-order logic and can be used to express both existentially and universally quantified constraints. However we believe CoLan provides better readability for domain experts compared to Galileo and other constraint programming languages such as the ILOG OPL language (Junker & Mailharro, 2003). Moreover CoLan was developed by one of our colleagues and we have the software to convert Colan into standard semantic web enabled XML CIF format. Also, Colan is mainly used in ConEditor+ as a declarative language for expressing constraints and not used for constraint programming. CoLan is converted into CIF which, in turn, is converted into a SPARQL query and a predicate in Prolog by the Designers' Workbench for constraint processing. In comparison to SWRL (Horrocks *et al.*, 2004), a semantic web rule language developed by the W3C, CIF can express fully quantified constraints. SWRL has now been extended to CIF/SWRL in order to express fully quantified constraints (McKenzie, Gray, & Preece, 2004).

10 Conclusions and Future Work

This paper describes a methodology to enable domain experts to capture and maintain constraints in an engineering design environment. An ontology is used to represent the domain knowledge and constraints are expressed against this ontology. The context is a system known as the Designers' Workbench that has been developed to automatically check if all the constraints have been satisfied and if not, enable the designers to resolve them. To function, the Designers' Workbench must be provided with a set of task specific requirements, and generic (company-wide) design constraints. Originally, the latter needed a knowledge engineer to study the design rule book(s), consult the design engineer (domain expert) and encode all the constraints into the Designers' Workbench's KB. We describe the tool ConEditor+ that has been developed to help domain experts themselves to capture and maintain engineering design constraints.

On the basis of the studies done in the domain of kite design and then in part of the Rolls-Royce domain, we have demonstrated the following aspects with the help of examples and experiments:

(i) An explicit representation of application conditions together with the constraints and the domain ontology could help a system in: a) reducing the number of inconsistencies and b) detecting subsumption, redundancy, fusion and suggesting appropriate refinements between pairs of constraints. In particular, we have demonstrated how inferencing from the domain ontology (using `owl:equivalentClass`, `owl:equivalentProperty`, `rdfs:subClassOf`) together with an explicit representation of application conditions and constraints could be used by a system to support the maintenance of constraints.

(ii) ConEditor+ is a useful system that enables domain experts to capture and maintain constraints. ConEditor+'s inferencing done using the domain ontology plays an important role in supporting the maintenance of constraints.

INSERT FIGURE 10 HERE

The proposed architecture that shows how ConEditor+ fits into a wider framework is given in Fig. 10. A Design Standards author initially inputs all the design rules (constraints) into ConEditor+. The design constraints are then converted into a standard machine interpretable format (CIF). CIF is then processed by the Designers' Workbench and converted into a SPARQL query and a Sicstus predicate. As can be seen from
Fig. 10, it is planned to interface the Designers' Workbench to a sophisticated knowledge-based engineering (KBE) system. The Designers' Workbench would then be called from the main system, the KBE, effectively as a sub-process to check the consistency of a design, or part of a design, produced by the KBE.

In fact, Fig. 10 only represents one aspect (the design rule book) of the knowledge which is both generated and used in a contemporary knowledge-based engineering firm which is involved in design, manufacturing & maintenance. For example, there are a number of further additional knowledge repositories needed by today's KBE systems, including:

- Design templates (and conditions under which they should be used, i.e. *application conditions*)
- Libraries of designs for components and their rationales
- Requirements and constraints of the various manufacturing environments
- Best practices as collected by several parts of the organization (including designers)
- Requirements and constraints mandated by the several organizations which service the engines

- Feedback from the servicing and maintenance organizations which indicate which problems actually arise in the field, some analysis of their possible causes, and suggested remedies.

The latter type of information is the focus of the IPAS project (www.3worlds.org), a DTI / Rolls-Royce funded project, which started in 2005. The last source of data quoted above makes it clear that there is an important Information Life Cycle inherent in the aero- industry, where information flows from Design to the Manufacturing units, and then to the Service / Maintenance facilities; the later in turn creates information which needs to be passed to designers so that future engines can be improved as a result of real-world feedback.

Fig. 11 shows this cycle:

INSERT FIGURE 11 HERE

It is also clear that there are vast amounts of data and information available from a variety of sources, and to make this information inter-operational, there is potentially a major role for ontologies as many of the data / information sources use different terminologies. This is certainly an important role for ontologies in the IPAS project. In fact in both the projects undertaken with Rolls-Royce (AKT and IPAS) we are not only using standard ontology maintenance procedures, but we are encountering many of the problems of contemporary ontology engineering, namely:

- ontology creation (seeking to develop ontologies systematically and to ensure that relevant aspects of trust and provenance are captured; deciding whether or not domain ontologies should be developed from high-level ontologies;
- ontology evolution (an ontology developed for one engine may need to be

modified so that it is applicable to a future engine) and,

- ontology modularization (for some services a sparse description of, say, the combustion chamber may be sufficient, but for other services much greater detail may be required).

Acknowledgements

This work is supported by the EPSRC Sponsored Advanced Knowledge Technologies project, GR/NI5764, which is an Interdisciplinary Research Collaboration involving the University of Aberdeen, the University of Edinburgh, the Open University, the University of Sheffield and the University of Southampton. We would like to acknowledge the assistance of engineers and designers in the Transmissions and Structures division of Rolls-Royce plc, Derby, UK. The IPAS project is funded by Rolls-Royce and the Department of Trade and Industry under the Technology Program, DTI Reference TP/2/IC/6/I/10292.

References

Ajit, S. (2008). *Capture and Maintenance of Constraints in Engineering Design.* PhD thesis (to appear), University of Aberdeen, Aberdeen, UK.

Ajit, S., Sleeman, D., Fowler, D. W., & Knott, D. (2004). ConEditor: Tool to Input and Maintain Constraints. In *Proceedings of the 14th International Conference on Engineering Knowledge in the Age of the Semantic Web, EKAW 2004* (pp. 466 - 468), Whittlebury Hall, Northampton, UK.

Ajit, S., Sleeman, D., Fowler, D. W., Knott, D., & Hui, K. (2005). Acquisition and Maintenance of Constraints in Engineering Design. In *Proceedings of the 3rd*

International Conference on Knowledge Capture, KCAP 2005 (pp. 173-174), Banff, Canada.

Ajit, S., Sleeman, D., Fowler, D. W., Knott, D., & Hui, K. (2007). ConEditor+: Capture and Maintenance of Constraints in Engineering Design. In *Proceedings of the IJCAI-07 Workshop on "Knowledge Management & Organizational Memories"* (pp. 6-11), Hyderabad, India.

Barker, V. E., & O'Connor, D. E. (1989). Expert Systems for Configuration at Digital: XCON and Beyond. *Communications of the ACM, 32*(3), 298-318.

Bassiliades, N., & Gray, P. (1995). CoLan: A Functional Constraint Language and Its Implementation. *Data and Knowledge Engineering, 14*(3), 203-249.

Bechhofer, S., van-Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., et al. (2004). *OWL Web Ontology Language Reference*. Retrieved 29 May 2008, from http://www.w3.org/TR/2004/REC-owl-ref-20040210/

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American, 284*(5), 28-37.

Borning, A., Maher, M., Martindale, A., & Wilson, M. (1989). Constraint Hierarchies and Logic Programming. In *Proceedings of the International Conference on Logic Programming (ICLP)* (pp. 149-164), Lisbon, Portugal.

Bowen, J., O'Grady, P., & Smith, L. (1990). A constraint programming language for Life-Cycle Engineering. *Artificial Intelligence in Engineering, 5*(4), 206-220.

Brickley, D., & Guha, R. V. (2004). *RDF Vocabulary Description Language 1.0: RDF Schema*. Retrieved 29 May 2008, from http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

Bultman, A., Kuipers, J., & Harmelen, F. V. (2000). Maintenance of KBS's by Domain Experts: The Holy Grail in Practice. In *Proceedings of the Thirteenth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE'00.*

Coenen, F. P. (1992). A Methodology for the Maintenance of Knowledge based Systems. In *Proceedings of the Niku-Lari, A. (Ed), EXPERSYS-92, IITT-International* (pp. 171-176), France.

Eden, M. (1998). *The Magnificient Book of Kites: Explorations in Design, Construction, Enjoyment and Flight*: Black Dog & Levanthal Publishers, New York.

Eriksson, H., Puerta, A., Gennari, J., Rothenfluh, T., Tu, S., & Musen, M. (1995). Custom-tailored development tools for knowledge-based systems. In *Proceedings of the Ninth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada.

Felfernig, A., Friedrich, G., Jannach, D., & Stumptner, M. (2004). Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence, 152*, 213-234.

Fowler, D. W., Sleeman, D., Wills, G., Lyon, T., & Knott, D. (2004). Designers' Workbench. In *Proceedings of the Twenty-fourth SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence* (pp. 209-221), Cambridge, UK.

Frayman, F., & Mittal, S. (1987). COSSACK: A constraints-based expert system for configuration tasks. In D. Sriram & R. A. Adey (Eds.), *Knowledge based Expert systems in Engineering: Planning and Design* (pp. 143-166).

Gray, P., Hui, K., & Preece, A. (2001). An Expressive Constraint Language for Semantic Web Applications. In *Proceedings of the E-Business and the Intelligent Web: Papers from the IJCAI-01 Workshop* (pp. 46-53), Seattle, USA.

Gray, P., & Kemp, G. (2006). Capturing Quantified Constraints in FOL, through Interaction with a Relationship Graph. In *Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2006)* (pp. 19-26), Podebrady, Czech Republic.

Gross, M., Ervin, S., Anderson, J., & Fleisher, A. (1987). Designing with constraints. In Y. E. Kalay (Ed.), *Computability of Design* (pp. 53-83).

Gruber, T. R. (1995). Towards Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies, 43*(5-6), 907-928.

Gruber, T. R., & Olsen, G. R. (1994). An Ontology for Engineering Mathematics. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Bonn, Germany.

Harary, F. (1962). A Graph Theoretic Approach to Matrix Inversion by Partitioning. In *Numerische Mathematik* (Vol. 4, pp. 128-135).

Harris, S., & Gibbins, N. (2003). 3store:Efficient Bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PSSS'03), International Semantic Web Conference*, Sanibel Island, Florida.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., & Dean, M. (2004). *SWRL: A Semantic Web Rule Language Combining OWL and RuleML.* Retrieved 01 June 2008, from http://www.w3.org/Submission/SWRL/

HP. (2000). *Helwett Packard Labs, Jena - A Semantic Web Framework for Java*. Retrieved 04 June 2007, from http://jena.sourceforge.net/

Junker, U., & Mailharro, D. (2003). The logic of ilog(j) configurator: Combining constraint programming with a description logic. In *Proceedings of the Proceedings of IJCAI'03 Workshop on Configuration*, Acapulco, Mexico.

Lin, L., & Chen, L. C. (2002). Constraints modelling in product design. *Journal of Engineering Design, 13*(3), 205-214.

McGuinness, D. L., & Harmelen, F. v. (2004). *OWL Web Ontology Language Overview, W3C Recommendation 10 February 2004*. Retrieved 29 August, 2006, from http://www.w3.org/TR/owl-features/

McKenzie, C., Gray, P., & Preece, A. (2004). Extending SWRL to Express Fully-Quantified Constraints. In *Proceedings of the Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML 2004), International Semantic Web Conference* (pp. 139-154), Hiroshima, Japan.

Noy, N. F., Fergerson, R. W., & Musen, M. A. (2000). The knowledge model of Protege-2000: Combining interoperability and flexibility. In *Proceedings of the International Conference on Knowledge Engineering and Knowledge Management (EKAW' 2000)*, Juan-les-Pins, France.

Prud'hommeaux, E., & Seaborne, A. (2007). *SPARQL Query Language for RDF, W3C Working Draft 26 March 2007*. Retrieved 04 June 2007, from http://www.w3.org/TR/rdf-sparql-query/

Serrano, D., & Gossard, D. (1992). Tools and Techniques for Conceptual Design. In C. Tong & D. Sriram (Eds.), *Artificial Intelligence in Engineering Design* (Vol. 1, pp. 71-116).

Soloway, E., Bachant, J., & Jensen, K. (1987). Assessing the Maintainability of XCON-in-RIME: Coping with Problems of a Very Large Rule-Base. In *Proceedings of the AAAI-87* (pp. 824-829), Seattle, USA.

Steward, D. V. (1962). On an Approach to Techniques for the Analysis of the Structure of Large Systems of Equations. In *SIAM Review* (Vol. 4, pp. 321-342).

Streeter, T. (1980). *The Art of the Japanese Kite*. Tokyo: Charles E Tuttle Company Inc.

Ullman, D. G. (2003). *The Mechanical Design Process*. New York: McGraw-Hill.

Wielinga, B., & Schreiber, G. (1997). Configuration-Design Problem Solving. *IEEE Expert, 12*(2), 49-57.

Yolen, W. (1976). *The Complete Book of Kites and Kite Flying*. New York: Simon and Schuster Trade.

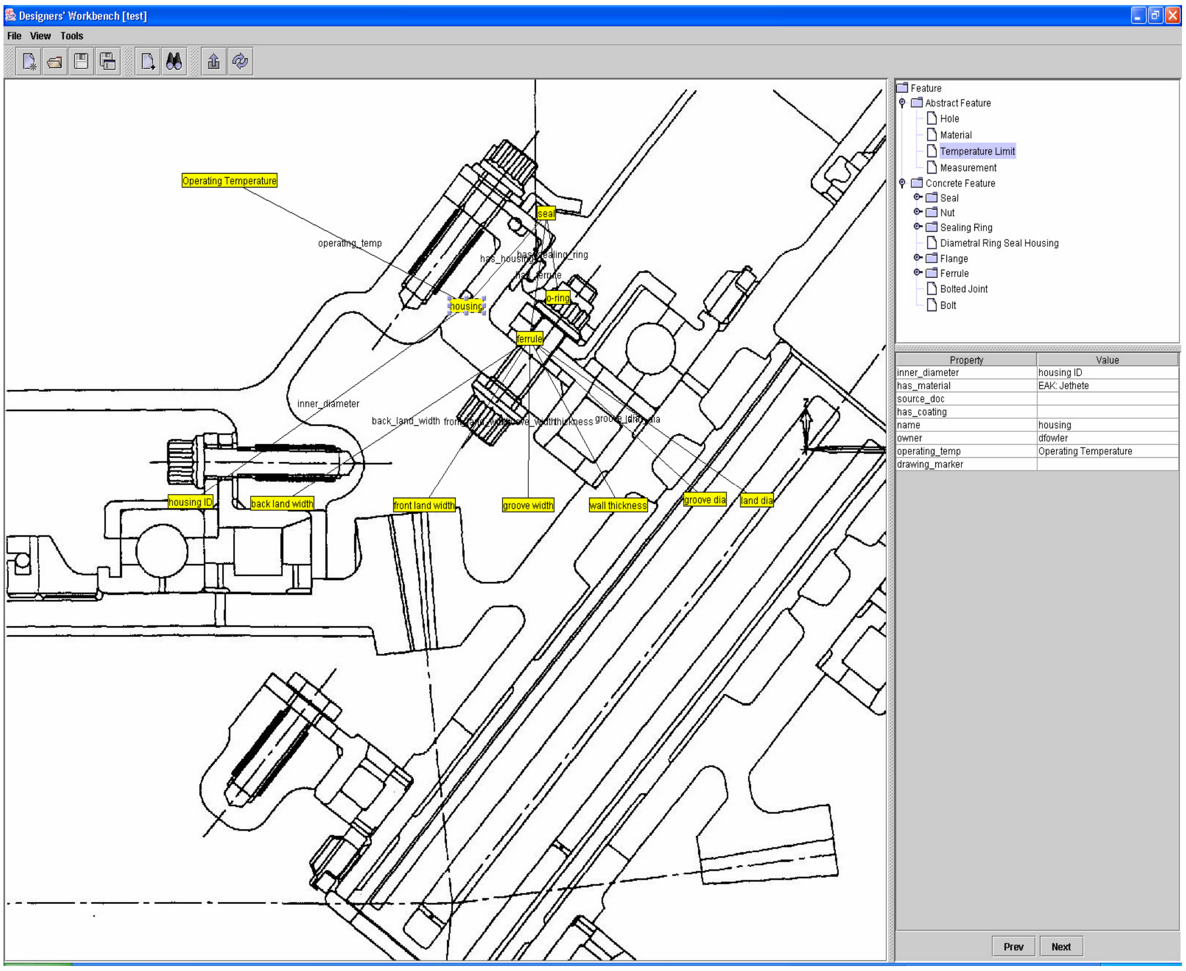
FIGURE CAPTIONS:

Fig. 1: A screenshot of the Designers' Workbench

Fig. 2: The class hierarchy of the jet engine ontology used with the Designers' Workbench (screenshot from the OWLViz plugin for Protégé)

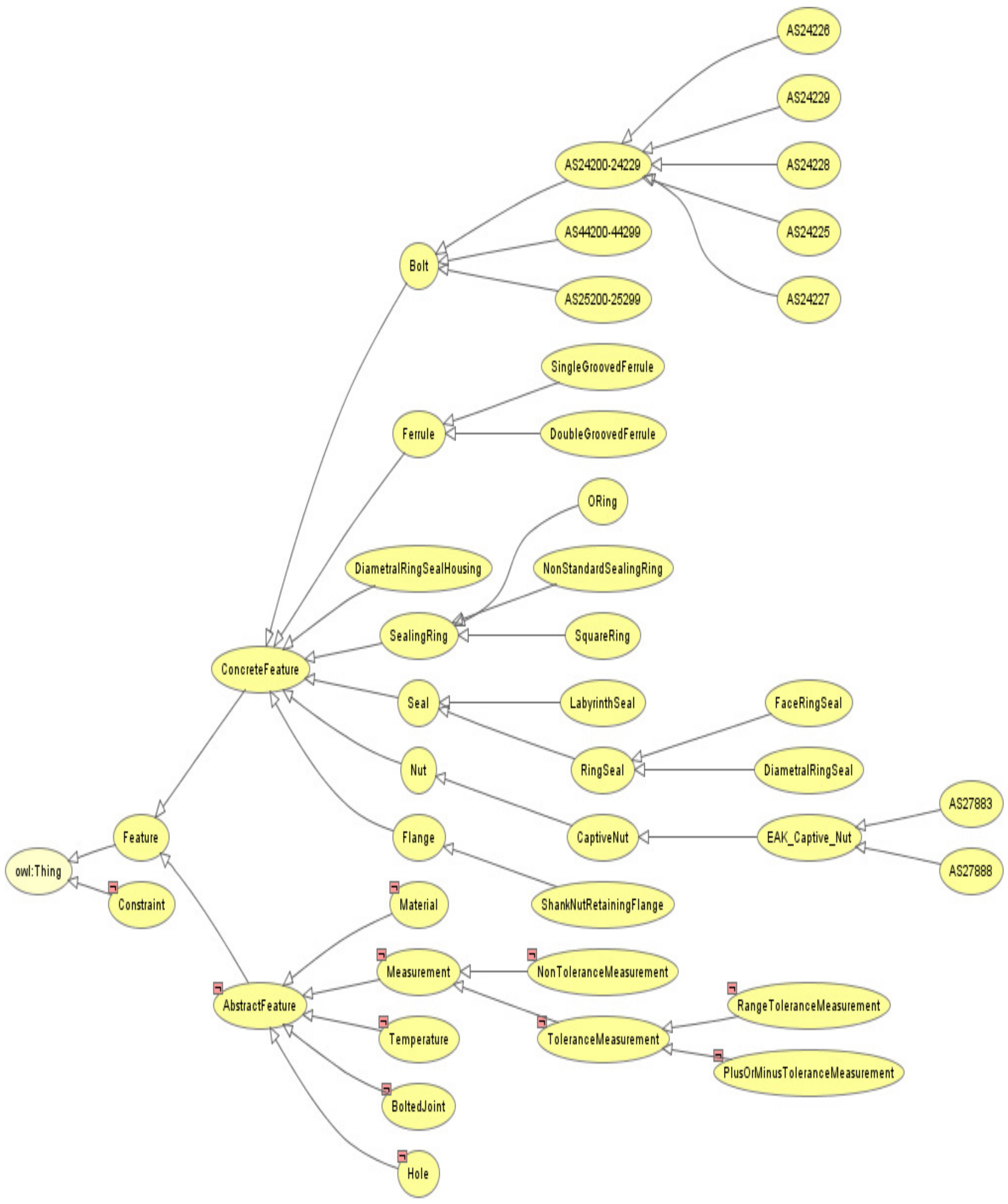
Fig. 3: The properties of the class DiametralRingSeal from the jet engine ontology (screenshot from Protégé)

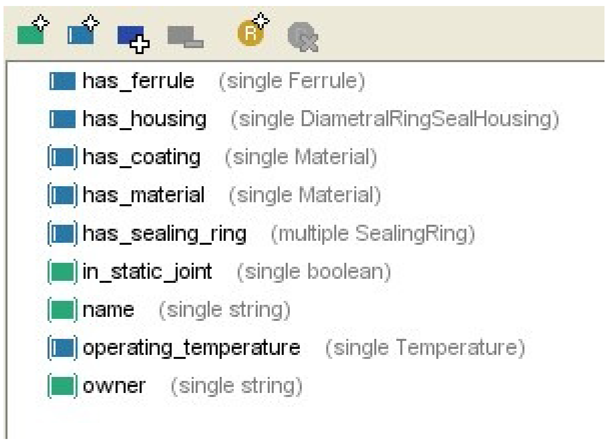
FIGURES:



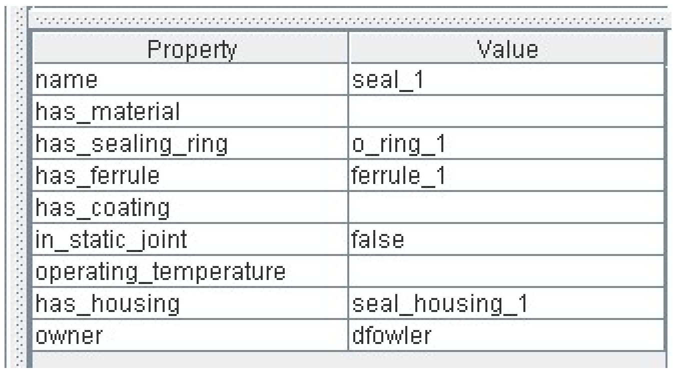
- Feature
  - Abstract Feature
    - Hole
    - Material
    - Temperature Limit
    - Measurement
  - Concrete Feature
    - Seal
    - Nut
    - Sealing Ring
    - Diametral Ring Seal Housing
    - Flange
    - Ferrule
    - Bolted Joint
    - Bolt

Property	Value
inner_diameter	housing ID
has_material	EAK: Jethete
source_doc	
has_coating	
name	housing
owner	dfowler
operating_temp	Operating Temperature
drawing_marker	


AS24226
AS24229
AS24200-24229
AS24228
AS44200-44299
AS24225
Bolt
AS25200-25299
AS24227
SingleGroovedFerrule
Ferrule
DoubleGroovedFerrule
ORing
DiametralRingSealHousing
NonStandardSealingRing
SealingRing
SquareRing
ConcreteFeature
Seal
LabyrinthSeal
FaceRingSeal
Nut
RingSeal
DiametralRingSeal
AS27883
Feature
Flange
CaptiveNut
EAK_Captive_Nut
owl:Thing
AS27888
Constraint
Material
ShankNutRetainingFlange
Measurement
NonToleranceMeasurement
AbstractFeature
RangeToleranceMeasurement
Temperature
ToleranceMeasurement
PlusOrMinusToleranceMeasurement
BoltedJoint
Hole



has_ferrule (single Ferrule)
has_housing (single DiametralRingSealHousing)
has_coating (single Material)
has_material (single Material)
has_sealing_ring (multiple SealingRing)
in_static_joint (single boolean)
name (single string)
operating_temperature (single Temperature)
owner (single string)



Property	Value
name	seal_1
has_material	
has_sealing_ring	o_ring_1
has_ferrule	ferrule_1
has_coating	
in_static_joint	false
operating_temperature	
has_housing	seal_housing_1
owner	dfowler

9.3.2 Internally trapped nuts (see Fig 4 Table 4)

TABLE 4

PCD ABOVE TO	2M
mm mm	mm
150 - 180	1,00
180 - 300	0,80
300	0,60

Ø N MIN. = PCD (NOM) + 2M + MAX. NUT WIDTH (SEE TABLE 5)

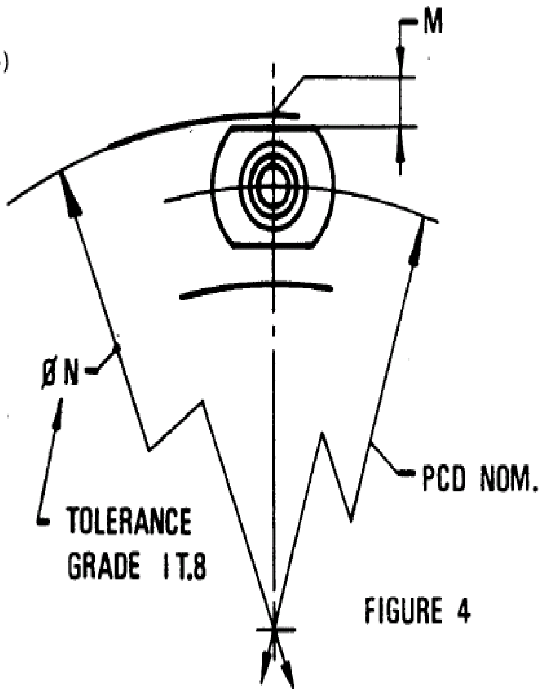
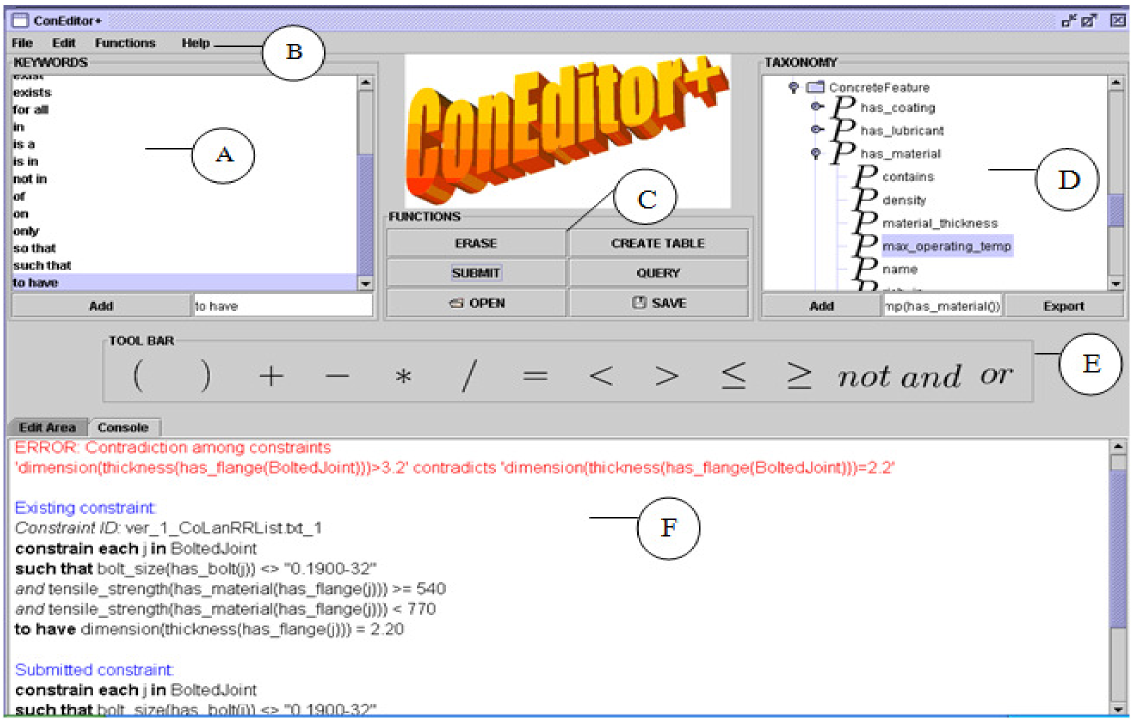


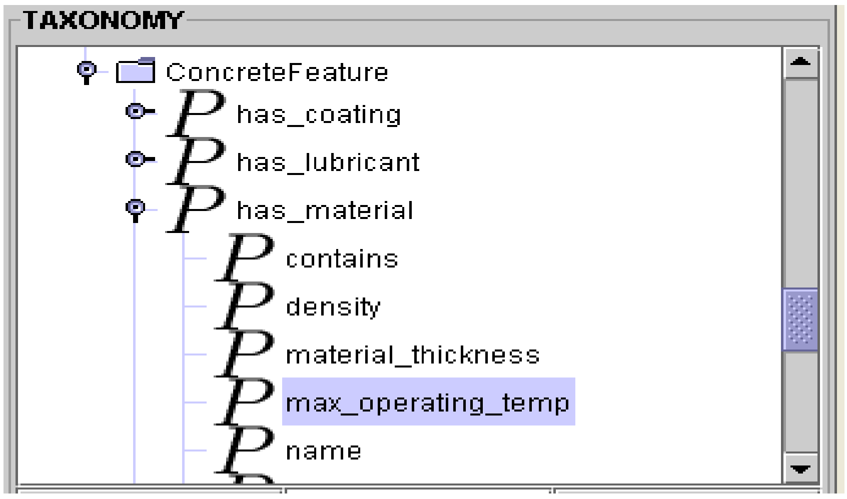
FIGURE 4

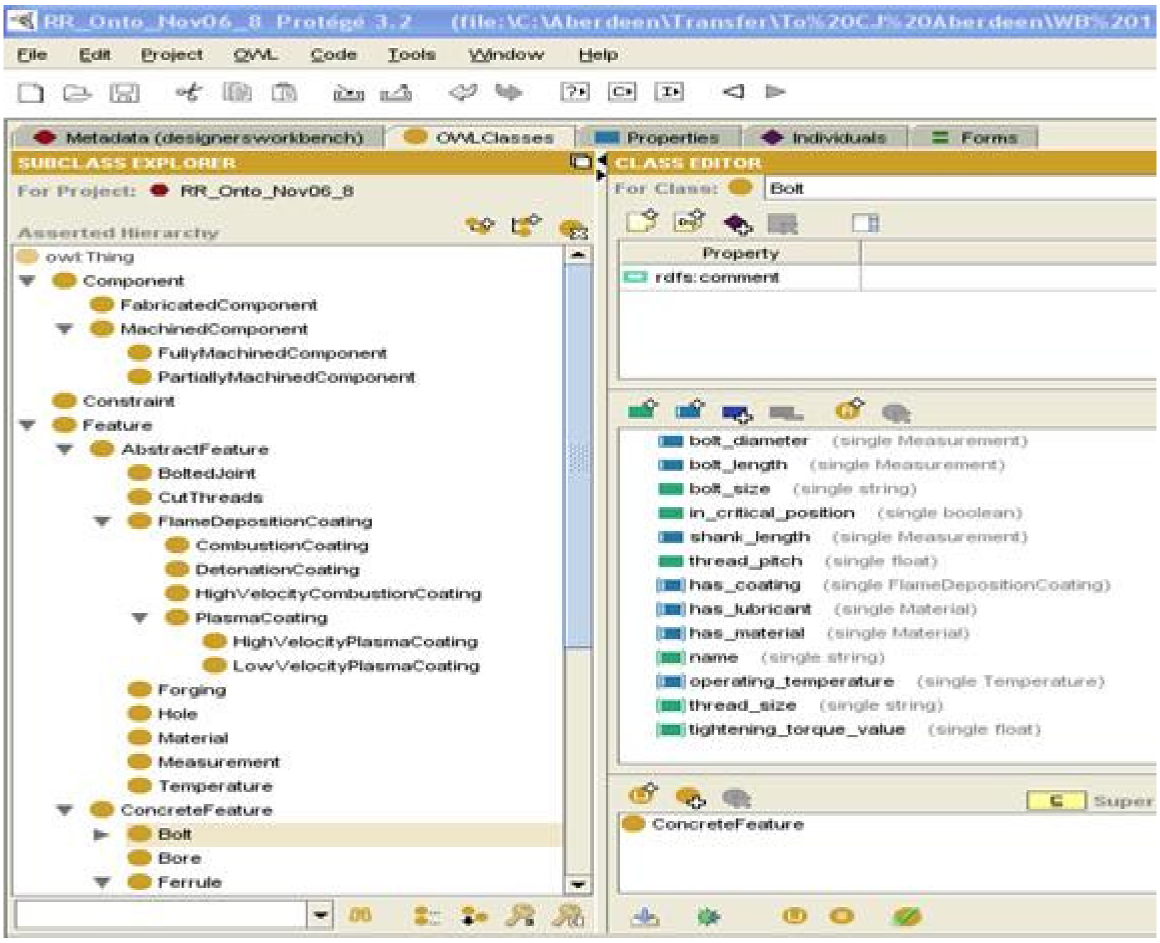


```
ERROR: Contradiction among constraints
'dimension(thickness(has_flange(BoltedJoint)))>3.2' contradicts 'dimension(thickness(has_flange(BoltedJoint)))=2.2'

Existing constraint:
Constraint ID: ver_1_CoLanRRList.txt_1
constrain each j in BoltedJoint
such that bolt_size(has_bolt(j)) <> "0.1900-32"
and tensile_strength(has_material(has_flange(j))) >= 540
and tensile_strength(has_material(has_flange(j))) < 770
to have dimension(thickness(has_flange(j))) = 2.20

Submitted constraint:
constrain each j in BoltedJoint
such that bolt_size(has_bolt(j)) <> "0.1900-32"
```


TAXONOMY
ConcreteFeature
has_coating
has_lubricant
has_material
contains
density
material_thickness
max_operating_temp
name


RR_Onto_Nov06_8 Protégé 3.2 (file:\C:\Aberdeen\Transfer\To%20CJ%20Aberdeen\WB%201
File Edit Project OWL Code Tools Window Help
Metadata (designersworkbench)
OWLClasses
Properties
Individuals
Forms
SUBCLASS EXPLORER
CLASS EDITOR
Super

For Project: RR_Onto_Nov06_8

Asserted Hierarchy

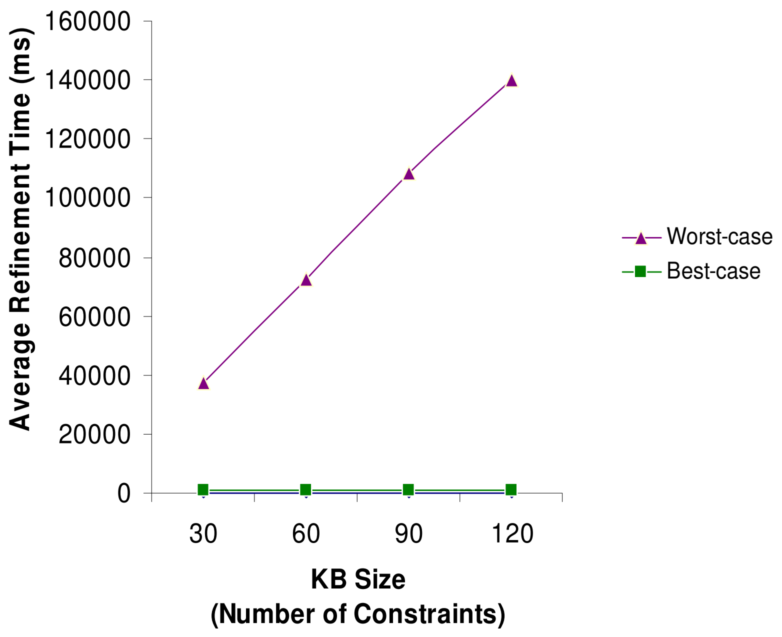
```
owl:Thing
▼ Component
    FabricatedComponent
  ▼ MachinedComponent
      FullyMachinedComponent
      PartiallyMachinedComponent
  Constraint
▼ Feature
  ▼ AbstractFeature
      BoltedJoint
      CutThreads
    ▼ FlameDepositionCoating
        CombustionCoating
        DetonationCoating
        HighVelocityCombustionCoating
      ▼ PlasmaCoating
          HighVelocityPlasmaCoating
          LowVelocityPlasmaCoating
      Forging
      Hole
      Material
      Measurement
      Temperature
  ▼ ConcreteFeature
    ► Bolt
      Bore
    ▼ Ferrule
```

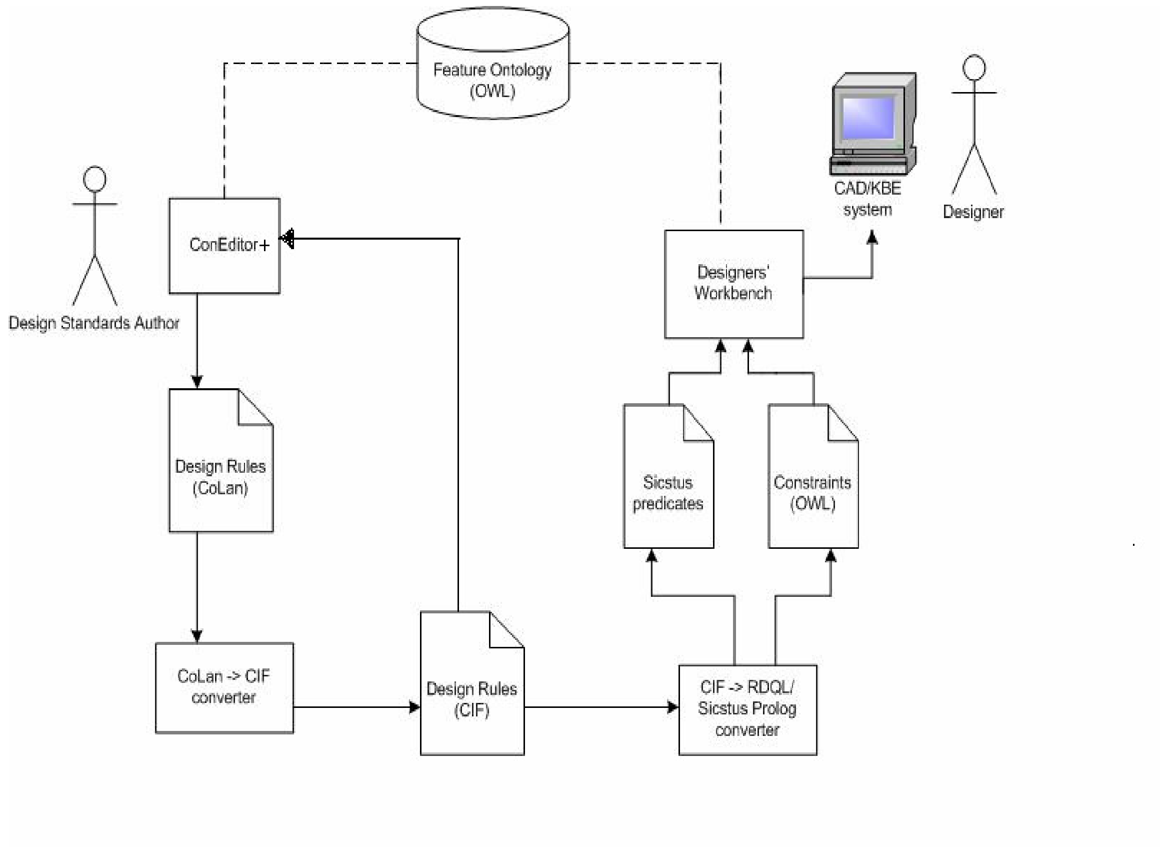
For Class: Bolt

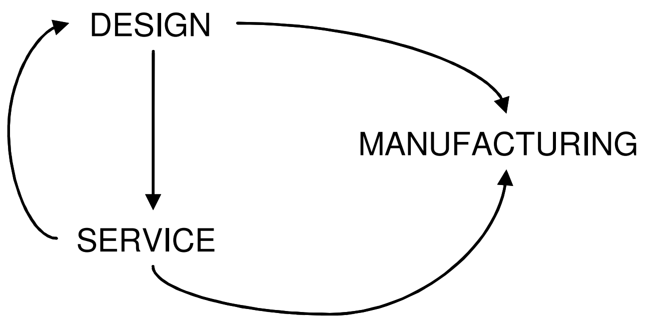
Property	
rdfs:comment	

```
bolt_diameter   (single Measurement)
bolt_length   (single Measurement)
bolt_size   (single string)
in_critical_position   (single boolean)
shank_length   (single Measurement)
thread_pitch   (single float)
has_coating   (single FlameDepositionCoating)
has_lubricant   (single Material)
has_material   (single Material)
name   (single string)
operating_temperature   (single Temperature)
thread_size   (single string)
tightening_torque_value   (single float)
```

ConcreteFeature


Average Refinement Time (ms)
160000
140000
120000
100000
80000
60000
40000
20000
0
30
60
90
120
KB Size
(Number of Constraints)
Worst-case
Best-case


Feature Ontology (OWL)
CAD/KBE system
Designer
ConEditor+
Design Standards Author
Designers' Workbench
Design Rules (CoLan)
Sicstus predicates
Constraints (OWL)
CoLan -> CIF converter
Design Rules (CIF)
CIF -> RDQL/ Sicstus Prolog converter


DESIGN
MANUFACTURING
SERVICE