J. Lin, M. S. Fox and T. Bilgic, (1996), "A Requirement Ontology for Engineering Design", Concurrent Engineering: Research and Applications , Vol. 4, No. 4, pp. 279-291, Sept. 1996.

A Requirement Ontology for Engineering Design

Jinxin Lin, Mark S. Fox and Taner Bilgic

Enterprise Integration Laboratory, Dept. of Industrial Engineering
University of Toronto, Toronto, Canada M5S 3G9
tel: +1-416-978-6823; fax: +1-416-971-2479; email: {jlin, msf, taner}@ie.utoronto.ca

August 1, 1996

Abstract

We present an ontology for representing requirements that supports a generic requirements management process in engineering design domain. The requirement ontology we propose is a part of a more general ontology to capture engineering design knowledge. Objects included in this general ontology are parts, features, requirements, and constraints. We define a generic requirements management process and raise issues that any requirement representation must address like communication, traceability, completeness, consistency, document creation, and managing change. We use first-order logic to define the objects and their attributes, and identify the axioms capturing the constraints and relationships among the objects. We show how the axioms can be used in answering the issues raised.

Key words: requirement representation, ontology, design knowledge, collaborative design, terminology and axioms.

1.0 Introduction

Requirements Management is one of the key elements that must be addressed by concurrent engineering (CE) [11]. Yet the representation and management of requirements is problematic in CE. Requirements are often ambiguous, incomplete and redundant in a CE environment. There is a lack of traceability of the requirements and insufficient decomposition of requirements [12]. Requirements generated by different members in a concurrent engineering team may be contradictory since different authors may have different perspectives on the system [16]. Authors of requirements use different terminology and hence the same term is applied to different concepts and different terms are used to denoted the same entity. Requirements are also changed frequently during the design process due to the changes of technology and customer's objectives [6]. Documents have to be maintained about requirements detailing: (i) decisions made on the scope of the requirements, (ii) resolution of ambiguous statements, and traceability links between requirements and the system specification and owners and approvers. Requirements are usually constructed in accordance with legislation and standards. We view the requirements management process one of creating, communicating, maintaining, and verifying requirements as in [3], however we take a formal approach in representing requirements.

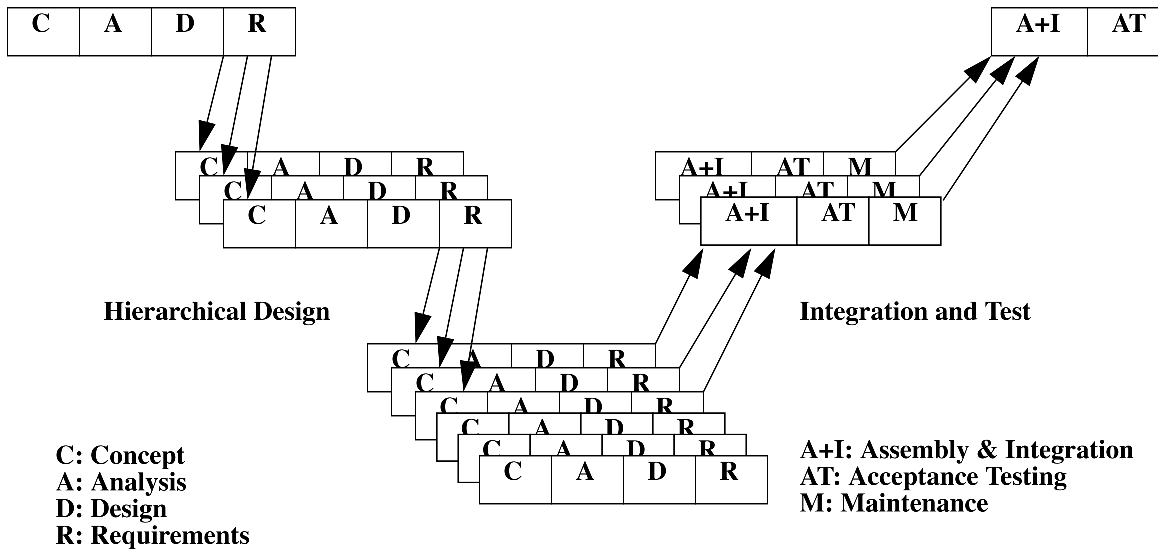
Although requirements and requirements management are encountered in many different facets of systems engineering (see e.g. [15] and the references therein), our focus in this paper is the engineering design domain which requires the services and collaboration of many engineers. We are developing a Knowledge Aided Design (KAD) system to address many issues that arise in that particular setting [1][9][10]. A major issue in concurrent engineering and collaborative design is the creation and maintenance of a suitable representation for design knowledge that will be shared by many design engineers. This knowledge includes many concepts such as component structure, features, parameters, constraints, requirements, and more. In this paper, we concentrate on requirements and propose an ontology that supports the requirement management process.

In Section 2.0 and Section 3.0, we identify the issues raised by the requirements management process and introduce a formal way of representing the underlying knowledge. We also transcribe the issues raised as competency questions (Section 3.1). In Section 4.0, we present the product ontology which complements the requirements ontology discussed in detail in Section 5.0. In Section 5.5, constraints and deductive rules are discussed. These notions, together with the ontology described in the earlier sections constitute tools for reasoning about requirements and hence addressing issues raised. In Section 6.0, we give some example queries to show how the issues are tackled within our formalism. Section 7.0 concludes the paper.

2.0 Motivation

In [6], Fox and Salustri present a model of systems engineering suitable for the design of complex artifacts. The artifact is usually composed of many sub-systems which in turn composed of other sub-systems or components. Customer requirements are decomposed into requirements for the various sub-systems. Then design is proposed for each sub-system and the relationships between customer requirements and the specifications of the sub-systems derived. At the same time the analysis and testing of the systems are defined. Figure 1 shows the V-model highlighting the decomposition and integration aspects of system design. The left arm of the V shows the decomposition of requirements, and the decomposition of concept, analysis and design driven by the decomposed requirements. The right arm shows the bottom-up integration & testing of the subassemblies and final assembly. Concurrency in the V-model is achieved in two ways: *vertically*, where lower levels on the left of the V are begun before higher level designs are completed, and *horizontally* where Assembly and Test is performed for each level of design before or during the elaboration of lower levels.

FIGURE 1. V-Model of System Design



The V-model for systems engineering management is complex. The degree of concurrency combined with the levels of refinement and composition make it particularly difficult to manage requirements.

The typical approach to defining requirements is rather informal at the start of the life cycle and it is assumed that elaborations on the higher level requirements will make them more precise. However, in the aerospace industry the requirements have been rigorously defined right from the very beginning mainly due to the strict legislations imposed by the customers. Furthermore, it is often the case that the business contracts are based on the requirements documents and payment is conditioned on the demonstration of the artifact to meet the requirements. Therefore, elicitation, elaboration, documentation, satisfaction, and traceability of requirements are of utmost importance.

In light of this and the problems discussed at the beginning of the introduction, there is a need for a representation of requirements for engineering design that:

- Provides an unambiguous and precise terminology such that each engineer can jointly understand and use in describing requirements.
- Is generic, reusable and easy to extend.
- Allows traceability of the requirements, with dependencies and relationships among the requirements captured and stored.
- Support the detection of redundant or conflicting requirements.
- Integrates requirements with parts, features, parameters and constraints.
- Facilitates document creation conforming to customer/company/government rules and regulations.
- Facilitates the change management process.

3.0 Ontology

An Ontology is a formal description of objects and their properties, relationships, constraints, and behaviors. As in [4], we are interested in a formal and rigorous approach to the representation of knowledge. Our approach is:

- To provide a terminology for design that can be shared by all the engineers involved. The engineers use the same terminology so that they can work in the same design and achieve high-level collaboration.
- To define the meaning of the terminology using first-order logic which gives a precise and unambiguous semantics for each term. The precision and unambiguity avoid possible conflicts and different interpretations by different engineers.
- To develop a set of axioms capturing definitions and constraints on the terminology to enable automatic deduction from the design knowledge. The axioms allow our system to answer design relevant questions using not only information explicitly represented in the

object model, but also what can be deduced from it. The axioms also allow integrity checking of the design knowledge, i.e. detecting invalid data in the database and avoiding updates introducing conflicts among the data and the object model of design.

Terminology (objects) included in our ontology include parts, features, requirements, and constraints. We identify axioms describing the constraints and relationships among the objects. We characterize the design activity as a process of constructing the objects and axioms in the ontology as well as evaluating the satisfaction of requirements and constraints by the product structure and parameter values.

In the KAD system, we have implemented the ontology as Prolog axioms and developed a WWW-based interface for engineers to pose queries to the ontology which will then be answered by the system through reasoning about the axioms. We also designed a symbology for depicting the objects of the ontology in a graphical context deployed in the engineering interface.

3.1 Competency of the Ontology

We follow the methodology for the design and evaluation of ontologies as described in [8]. We first identify the problems in the domain that the ontology is trying to address. These problems are given in the forms of questions which the ontology is intended to answer. This group of questions is called *competency* of the ontology. The competency questions provide a characterization and justification for our approach and enable people to understand the scope and limitations of the approach. We have mentioned that we will develop a set of axioms (microtheory) to capture definitions and constraints on the terminology in the ontology. This microtheory must contain a necessary and sufficient set of axioms to represent and solve these questions, thus providing a declarative semantics for the system.

The following are some categories of competency questions supported by our ontology. In each category we list several typical questions.

- **Requirement refinement:**

  1. Do the decomposed requirements preserve the meaning of the original requirements?

  2. Are there redundant requirements generated during the requirement decomposition process?

  3. Is a requirement an explicit statement of the customer?

- **Requirement traceability:**

  1. Does this requirement decompose to others?

  2. What is the source of the requirement?

  3. Who posted the requirement?

  4. Does the requirement apply internally within a particular design team or externally?

- **Requirement satisfaction:**

1. Is a requirement satisfiable?
2. Are two (or more) requirements conflicting each other?
3. Which requirements are satisfied and which are violated?

- **Version & Change:**

1. Is this the latest version of the requirement?
2. What are the older versions of this requirement?
3. Does a change affect consistency of the requirements set?

- **Relationships of requirements to parts:**

1. Does a requirement impose a constraint on the part?
2. What are the requirements of a part?
3. On which parameter is the constraint imposed?

- **Product structure:**

1. What are the components of a part?
2. What features does the part have?
3. What constraints that the part must satisfy?
4. What constraints that parameter-X of the part must satisfy?
5. Where a specific type (or class) of parts are used?
6. What are components of Assembly P that exceed a certain percentage of the total mass?

4.0 Product Ontology

In this section we describe the portion of our product ontology used by our requirements ontology. This ontology extends the product representation used in [5].

Following the object-oriented tradition, in our ontology each object is associated with a unique name (which can be thought of as its ID). There are two types of objects: *class* (objects) and *instance* (objects). A class is used for representing a generalized type or category of object, and instance for a specific member of a class. Among many others, there are classes called *part, feature, requirement,* and *constraint* which classify the design objects. Each of these classes can be further divided into subclasses. We denote the subclass relationship by the predicate *subclassOf(.,.)*. For example, the following says that *eil_part* is a subclass of the class part: *subclassOf(eil_part, part)*.

An instance and a class are related by the predicated *instanceOf(.,.)*. For example, the following says that a particular part, *PRT131,* is an instance of the class *eil_part*: *instanceOf(-PRT131, eil_part)*.

Throughout this paper, we denote variables by lower case letters and constants by upper case letters. We use *p, f, r, c* (with or without subscripts) to denote variables of the classes part, feature, requirement, and constraint respectively.

4.1 Parts

A *part* is a component of the artifact being designed. The artifact itself is also viewed as a part. The concept of part introduced here represents a physical identity of the artifact, software components and services. Throughout the paper, for simplicity we use a small example to illustrate the concepts from the ontologies we develop. Nevertheless, the concepts do apply to more sophisticated engineering design domains. In this example, we assume that we are designing a desk spot lamp.

The structure of a part is defined in terms of the hierarchy of its component parts. The relationship between a part and its components is captured by the predicate *component_of*. The *component_of* is similar to the *subcomponent-of*, the *composed-of*, and the *part-of* relations defined in [7], [13], and [5] respectively, except that the axioms are made explicitly.

Between two parts *p* and *p'*, *component_of(p, p')* means that *p* is a component (subpart) of *p'*. For example, there are three components of a desk spot lamp, namely *Heavy_base, Small_head* and *Short_arm*:

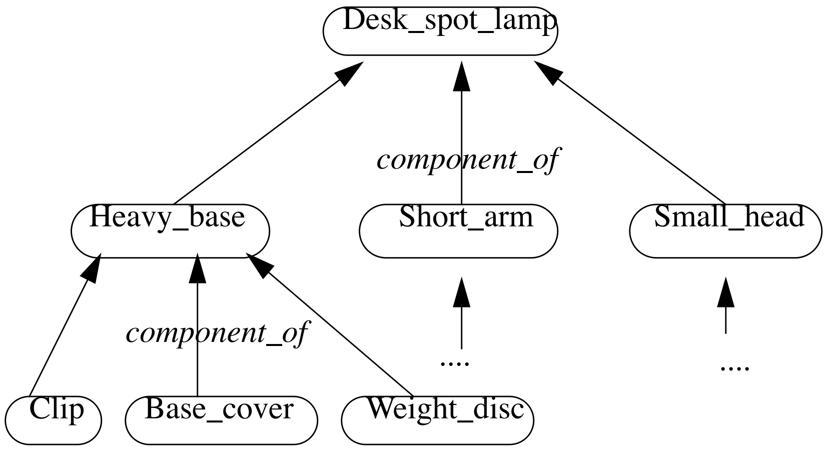
component_of(Heavy_base, Desk_spot_lamp).

component_of(Short_arm, Desk_spot_lamp).

component_of(Small_head, Desk_spot_lamp).

...

FIGURE 2. Components of Desk Spot Lamp



The relation *component_of* is transitive; that is, if a part is a component of another part that is a component of a third part then the first part is a component of the third part.

$(\forall\, p_1, p_2, p_3)\ component\_of(p_1, p_2) \wedge component\_of(p_2, p_3) \supset component\_of(p_1, p_3).$

The following two axioms state that a part cannot be a component of itself, and it is never the case that a part is a component of another part which in turn is a component of the first part. This shows that the relation *component_of* is non-reflexive and anti-symmetric:

$(\forall\, p)\ \neg\, component\_of(p, p).$

$(\forall\, p_1, p_2)\ component\_of\,(p_1, p_2) \supset \neg\, component\_of(p_2, p_1).$

A part can be a (sub-)component of another part. But since each part has a unique ID (its name), it cannot be a sub-component of two or more distinct parts that are not components of each other:

$(\forall\, p_1, p_2, p_3)\ component\_of(p_1, p_2) \wedge component\_of(p_1, p_3) \supset p_2 = p_3 \vee component\_of(p_2, p_3) \vee component\_of(p_3, p_2).$

The above four axioms guarantee that the part structure is in the form of *forest* consisting of one or more trees of parts.

The *component_of* relation relates objects lower in the component tree to the objects higher. By the relation it is possible to traverse upward in the component tree. There is often a need to traverse downward in the component tree, by introducing the relation *has_component*, which is defined as the inverse relation of *component_of*.

$$(\forall\, p_1, p_2)\ has\_component(p_1, p_2) \equiv component\_of(p_2, p_1). \qquad (1)$$

Parts can be made from the same model and be identical copies, and can be used as different assemblies. Then they are treated as different instances of the same class and associated with different ID. For example if we want to talk about two clips, say $Clip_1$ and $Clip_2$, are of the same kind, we can create a class called *Clip* and say that $Clip_1$ and $Clip_2$ are instances of this same class by the following terms:

$instanceOf(Clip_1, Clip),$

$instanceOf(Clip_2, Clip).$

Parts are classified into two types, depending upon the *component_of* relationship it has with the other parts in the hierarchy. The two types are: *primitive* and *composite*.

- A primitive part is a part that cannot be further subdivided into components. This type of parts exist at the lowest level of the artifact decomposition hierarchy. Therefore, a primitive part cannot have sub-components.

  $(\forall p)\ primitive(p) \equiv \neg\, (\exists\, p')\ component\_of(p', p)$

  Primitive parts serve as a connection between the design stage and the manufacturing stage.

- A composite part is a composition of one or more other parts. A composite part cannot be a leaf node in the part hierarchy; thus any part that is composite is not primitive.

  $(\forall\, p)\ composite(p) \equiv \neg\, primitive(p).$

Most composite parts are *assemblies,* which are composed of at least *two or more* parts.

$(\forall\ p)\ assembly(p) \equiv (\exists\ p_1, p_2)\ component\_of(p_1, p) \wedge component\_of(p_2, p) \wedge p_1 \neq p_2.$

Sometimes a designer may need to find out the *direct components* of a part. A part is a direct component of another part if there is no middle part between the two in the product hierarchy.

$(\forall\ p_1, p_2)\ direct\_component\_of(p_1, p_2) \equiv component\_of(p_1, p_2) \wedge \neg (\exists\ p')\ component\_of(p_1, p') \wedge component\_of(p', p_2).$

That is, p_1 is a direct component of p_2 if p_1 is a component of p_2 and there is no p' such that p_1 is a component of p' and p' is a component of p_2.

4.2 Features

There are different kinds of features associated with a part, e.g., geometrical features, functional features, assembly features, mating features, physical features, etc. [2][14]. We focus on geometrical and functional features. Examples of geometrical features are hole, slot, channel, groove, boss, pad, etc.; these are also called *form features*. Designers usually have in mind the purposes that they want these features to serve. For example, a designer introduces a hole as a feature to the arm of a desk spot lamp so that an electrical cord can run through it. Functional features describe the functionality of a part; they define what the part can be used for.

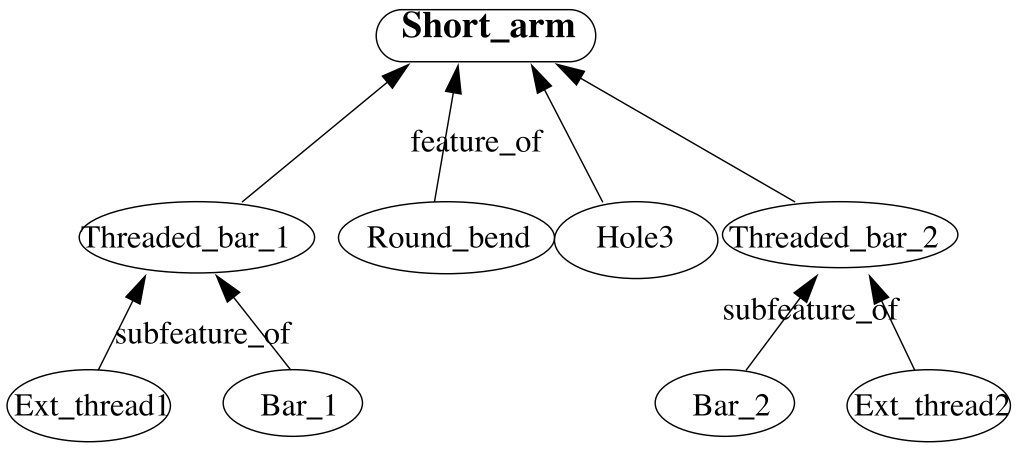
A part and its features are related by the predicate *feature_of(f, p)*, saying that *f* is a feature of part *p*. The following term represents the fact that a hole feature, called *Hole3*, is introduced in the short arm of the desk spot lamp:

feature_of(Hole3, Short_arm).

There can be *composite features* that are composed of several sub-features. For example, a threaded hole is a feature, which can be a component of a group of threaded holes that form a mounting feature. The term $subfeature\_of(f_1, f_2)$ says that feature f_1 is a subfeature of f_2.

Figure 3 shows the part Short_Arm and its features.

FIGURE 3. Features of Short Arm



It has the following *subfeature_of* terms.

subfeature_of (Ext_thread1, Threaded_bar_1).

subfeature_of (Bar_1, Threaded_bar_1).

subfeature_of (Bar_2, Threaded_bar_2).

subfeature_of (Ext_thread2, Threaded_bar_2).

The following axiom says that a subfeature of a feature of a part is also a feature of the part:

$(\forall f_1, f_2, p)\ subfeature\_of(f_1, f_2) \wedge feature\_of(f_2, p) \supset feature\_of(f_1, p).$

The *feature_of* and *subfeature_of* have inverse relations *has_feature* and *has_subfeature* defined in terms of axioms similar to (1), which we omit here.

4.3 Parameters

Parts can have parameters that define their properties such as weight, color, diameter, material, surface finish, etc. So can features (but the parameters must be meaningful with respect to features), e.g. a hole feature has a parameter representing the diameter of a hole. Parameters are denoted by functions in the first-order logic; for instance, the color of the *Short_arm* is denoted by *color(Short_arm)*.

Information about a parameter such as its *type* (string, integer, real number, boolean, etc.), *unit of measurement* (pound, kg, liter, etc.), and other related information need to be recorded. This is represented by the predicates:

type(para_name, part_or_feature, type),

unit(para_name, part_or_feature, unit),

...

For example, type and unit information of a weight parameter of any part *p* is defined as:

type(Weight, p, Real),
unit(Weight, p, Pound).

And for a diameter parameter of a hole, it is:

type(Diameter, Hole3, Real),
unit(Diameter, Hole3, Centimeter). ("cm" as the shorthand)

A parameter is associated with a domain defining the values that the parameter can obtain. In our framework, we specify this by the a domain constraint discussed in a later section.

4.4 Version

Design is an evolutionary process during which changes occur frequently. Before reaching its maturity each object of requirements, parts, features, and constraints may undergo many transformations and revisions. Versions of the objects are created to record the history of the design. We regard each version itself to be an object, which in our case is one of requirements, parts, features, or constraints. Version history is recorded by the predicate: *derived_-from(o,o')* meaning that object *o* (a version) is derived from object *o'* (another version). Each version has a time at which the version is created. The term *creation_time(o)* denotes the creation time of a version *o*. In a subsequent report[1], we will describe the details of the aspect of change and version management in our ontology.

5.0 Requirements

Requirements specify the properties (functional, structural, physical, etc.) of the artifact being designed. Initial requirements often come from the customer representing his/her wishes. These can be vague and incomplete (in some case, even inconsistent). A process in design is then to clarify or interpret the customer's wishes into more concrete objectives through consultations between the designer and the customer. In this process the initial requirements are decomposed into sub-requirements carrying greater details of the specification of the artifact.

5.1 Decomposition of Requirements

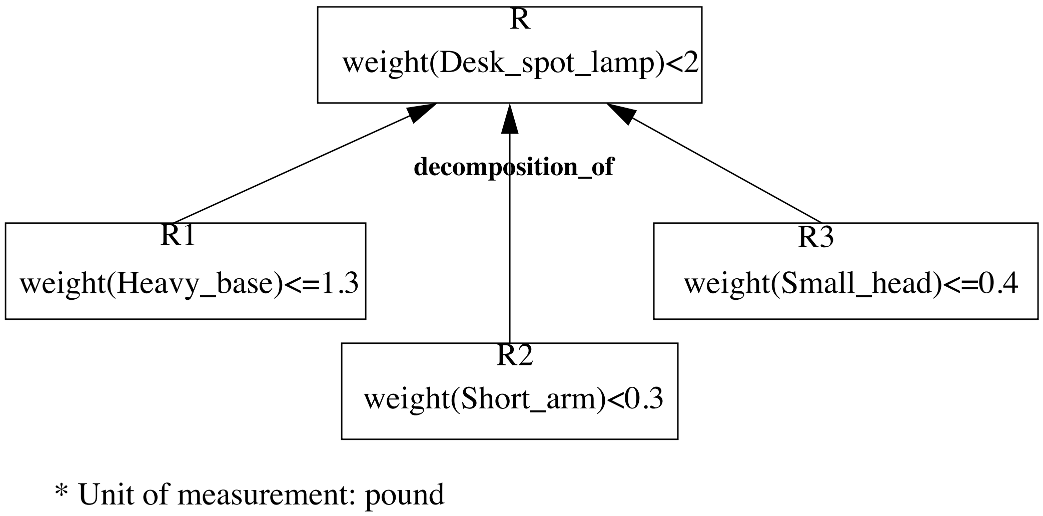
The hierarchy of requirements is built on the relation *decomposition_of*. Figure 4 shows the decomposition of the weight requirement for the desk spot lamp. (Weight factor is particularly important in designing equipments for aerospace usage, where a weight requirement of a part is often decomposed into sub-requirements on the components of the part.)

1. In preparation.

This relation *decomposition_of*, like the *component_of*, should be transitive, anti-symmetric and non-reflexive. These axioms are similar to that of part and we do not repeat them here. The hierarchy is a single or multiple tree structure, with the roots of the trees being requirements originated from the customer. We can also talk about *direct decomposition* of a requirement, and *primitive requirements*. The definitions are again similar to that of part, e.g. the primitive requirements are defined as the leaf requirements in a tree of the decomposition hierarchy:

$(\forall\ r)\ primitive(r) \equiv \neg\ (\exists\ r')\ decomposition\_of(r', r).$

FIGURE 4. Weight Requirement Decomposition



Every requirement is associated with an *expression* describing in logical form the content of the requirement. Let *req_exp(r)* denote the expression of requirement *r*. Primitive requirements are detailed specifications on properties of the artifact. Their logical expressions often involve some particular parameters of one or more parts. For example, the following defines the requirement (of the name, say, *R*) "The weight of the desk spot lamp must be within 2.0 0.1 pound".

$$req\_exp(R) \equiv 1.9 \leq weight(Desk\_spot\_lamp) \leq 2.1. \qquad (2)$$

Note in this requirement the unit of measurement for weight (which is *pound*) agrees with the unit of measurement information of the weight parameter. If the two are different, e.g. the unit of measurement of the weight parameter is *kilogram* (i.e. we have *unit(Weight, Desk_spot_lamp,Kilogram)*), then pound must be converted to kilogram. Assuming that the function *pound_to_kg* does the job, we have the requirement expression:

$$req\_exp(R) \equiv pound\_to\_kg(1.9) \leq weight(Desk\_spot\_lamp) \leq pound\_to\_kg(2.1).$$

For conversion to/from SI units, functions like *pound_to_kg* can be used which must be part of any engineering design ontology. In the following we assume that no disparity exists on the unit of measurement for each parameter so that no conversion is needed.

The expression of a high level requirement can be defined explicitly as a logical sentence (similarly to the primitive requirements above) or defined in terms of lower level requirements. The latter usually occurs when a higher level requirement consists of several lower level ones and it is simply an aggregation of requirements. For instance,

R: Motor safety requirement

consists of the following two sub-requirements:

R1: There should be an emergency switch to stop the running of the motor.

R2: There should be a surface cover for the motor.

Then we have

$$req\_exp(R) \equiv req\_exp(R1) \wedge req\_exp(R2).$$

In this case, the *decomposition_of* relation can be understood as a simple "consist of" relation.

The decomposition process must ensure that the meaning of the original requirement be preserved. Suppose $r_1,..., r_n$ are the direct decompositions of r. Then it must be the case that

$$req\_exp(r_1) \wedge ... \wedge req\_exp(r_n) \supset req\_exp(r). \qquad (3)$$

That is, if the lower level requirements are satisfied then the higher level one is also. The converse may not be true, i.e., it may not be the case that

$$req\_exp(r) \supset req\_exp(r_1) \wedge ... \wedge req\_exp(r_n). \qquad (4)$$

This applies to all *derived* requirements, a notion we will formally discuss in the next section. Figure 4 shows an example of derived requirements, where nothing in requirement R mentions R1, R2 and R3, yet the later three are decompositions of R. In this case, we say that R1, R2 and R3 are *derived* requirements (derived from R). We can see that (3) holds but (4) does not.

If indeed both (3) and (4) are true, we say that the decomposition step is *faithful*, i.e., the sub-requirements give an *exact interpretation* of the original requirement. The sub-requirements are also called *faithful decompositions* of the original requirement. Faithful decomposition is desirable, since we want the original customer's requirements being observed as much as possible. However, it may be difficult to achieve.

5.2 Derived and Explicit Requirements

We have seen an example of explicit and derived requirement in the last section. In this section we will formally discuss these two notions. Let *explicit(r)* mean that *r* is an explicit requirement and *derived(r)* mean *r* is a derived requirement.

A requirement is *explicit* if it is given by the customer or is a faithful decomposition of a customer's requirement.

Then the requirements at the top of the decomposition hierarchy are explicit since the initial requirements come from the customer:

$(\forall\ r)\ \neg(\exists\ r')\ decomposition\_of(r, r') \supset explicit(r).$

If a requirement is explicit then all of its ancestors are explicit:

$(\forall\ r, r')\ explicit(r) \wedge decomposition\_of(r, r') \supset explicit(r').$

A requirement is *derived* if it is not explicit. Every derived requirement has a parent from which the requirement is derived:

$(\forall\ r)\ derived(r) \supset (\exists\ r')\ decomposition\_of(r, r').$

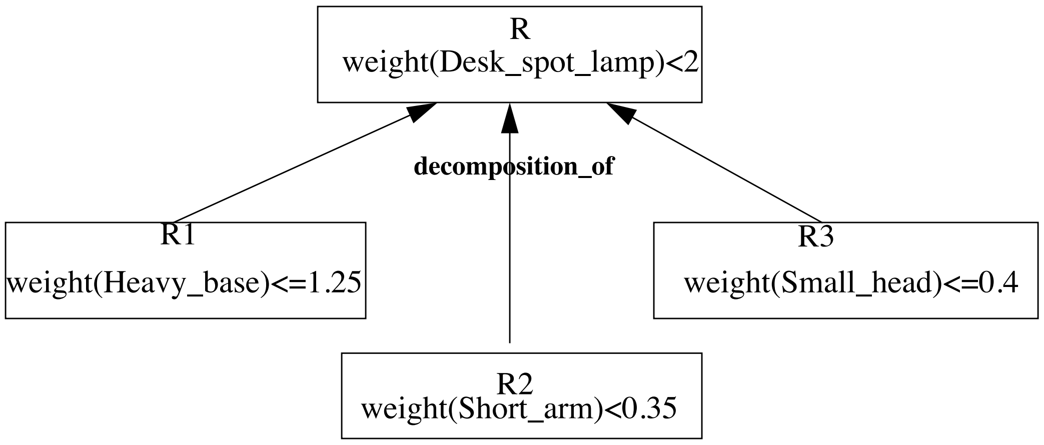
If a requirement is derived then all of its decompositions are derived:

$(\forall\ r, r')\ derived(r) \wedge decomposition\_of(r', r) \supset derived(r').$

So for a requirement that has a parent (i.e. not a root of a tree in the decomposition hierarchy), it is explicit if its parent (single) is explicit and the decomposition from the parent to the child(-ren) is faithful, and it is derived if its parent is derived or its parent is explicit but the decomposition from the parent to the child(-ren) is not faithful.

Derived requirements are subjected to changes during the design process, but explicit requirements are not (without negotiation with the customer). In the example shown in Figure 4, suppose in the design process R2 is found difficult to meet (i.e. Short arm weighs over 0.3 pound) while R1 and R3 are fine. A process is then triggered to revise R1, R2 and R3, reallocating the weights to that such as depicted in Figure 5.

FIGURE 5. Derived New Weight Requirement



* Unit of measurement: pound

5.3 Requirement Source

A requirement can also be distinguished as *external* or *internal*, depending on where the requirement originated.

- **External requirements** are specified in a design project external to the design team, often the customer. Modification to these requirements requires higher level approval (e.g. negotiation with the customer), and hence is not under discretion of the designer. Let *customer(a)* denote that the agent *a* is the customer (the concept of customer is very general in the sense that it can be a single person, a group of people or an organization), and *req_posted_by(r, a)* denote that the requirement *r* is posted by the agent *a*. Then all external requirements are posted by customers, i.e. agents external to the design team.

  $(\forall\ r)\ external(r) \equiv (\exists\ a)\ req\_posted\_by(r, a) \wedge customer(a).$ (5)

  If a requirement is external, then it is an explicit requirement:

  $(\forall\ r)\ external(r) \supset explicit(r).$ (6)

- **Internal requirements** are those posted by the design team members internally. These requirements may originate as a result of the decomposition of external requirements. They are temporary in nature and often subjected to changes during the design process. Let *design_team_member(a)* denote that the agent *a* is a member of the design team.

  $(\forall\ r)\ internal(r) \equiv (\exists\ a)\ req\_posted\_by(r, a) \wedge design\_team\_member(a).$

It is important to know whether a requirement is external or internal. A requirement originated external to the design team need to be dealt differently than a requirement originated internal to the team. This information is very useful when there is a violation of the requirements, and so some of them need to be relaxed or modified.

The *source* of a requirement is the top level requirement from which the current requirement is decomposed. Let *req_source*(r_1, r_2) denote that r_2 is the source of r_1.

The following axiom defines the source of a requirement in terms of the decomposition hierarchy.

$(\forall\ r_1, r_2)\ req\_source(r_1, r_2) \equiv decomposition\_of(r_1, r_2) \wedge \neg(\exists\ r)\ decomposition\_of(r_2, r).$

Requirements can subsume each other. The subsume relation is:

$(\forall\ r_1, r_2)\ subsume(r_1, r_2) \equiv [req\_exp(r_1) \supset req\_exp(r_2)].$ (7)

Thus if r_1 subsumes r_2 and both r_1 and r_2 are decompositions of some other requirement *r*, then r_2 can be deleted from the decomposition hierarchy of *r* without affecting the meaning of *r*. The subsume relation can be used to determine redundancy in the requirement decomposition process.

With the subsume relation, we can give a property of faithful decomposition. If r_1,..., r_n are faithful decompositions of *r*, then we have:

$subsume(r, r_1) \wedge ... \wedge subsume(r, r_n)$.

This property is easily derived from (4).

5.4 Several Classes of Requirements

Requirements can be classified into physical, structural, functional, cost, performance requirements, (and many others), depending on the properties of the artifact that the requirements concern with. Below we list some examples of the requirements and their logical descriptions.

1. *Physical requirements* are requirements related to "physical" properties of the artifact such as weight, height, color, material, stiffness, power consumption, etc. In the desk spot lamp example, we may have the requirement "The weight of a desk spot lamp must be within 2.0 ± 0.1 pound".

   $$req\_exp(R) \equiv (\forall\ p)\ desk\_spot\_lamp(p) \supset 1.9 \leq weight(p) \leq 2.1. \quad (8)$$

   Note that this logical sentence is essentially the same as (2) except that here we assume the exact name of the artifact is unknown while in (2) it is known.

2. *Structural requirements* are requirements about decomposition of the artifact into subparts and the topological arrangement of them, or requirements about form features of the artifact. For example, the customer may specify that the desk spot lamp must consist of a base, a head and an arm. This can be described as the logical sentence:

   $req\_exp(R) \equiv (\forall\ p)\ desk\_spot\_lamp(p) \supset (\exists\ p_1, p_2, p_3)\ base(p_1) \wedge head(p_2) \wedge arm(p_3) \wedge component\_of(p_1, p) \wedge component\_of(p_2, p) \wedge component\_of(p_3, p)$.

   Some structural requirements may be about form features of the artifact, e.g. "the short arm must have a hole of diameter ranging between 1 ± 0.5 cm so that an electrical cord can run through it".

   $req\_exp(R) \equiv (\forall\ p)\ arm(p) \supset (\exists\ f)\ hole\_feature(f) \wedge feature\_of(f, p) \wedge\ 0.5 \leq diameter(f) \leq 1.5$.

3. *Performance requirements* specify the performance goals for the artifact. The following is an example of the performance requirement "The artifact (desk spot lamp) should be able to illuminate more than half a square meter of room":

   $req\_exp(R) \equiv (\forall\ p)\ desk\_spot\_lamp(p) \supset (\exists\ f)\ illuminating\_feature(f) \wedge feature\_of(f, p) \wedge illumination\_area(f) \geq 0.5$.

   This says that the desk spot lamp should have a feature (called "illuminating") associated with a "illumination_area" whose value is greater than 0.5 (square meter).

4. *Functional requirements* specify functional properties of the artifact. A designer usually introduces functional features to the artifact in response to functional requirements. The following is an example of the functional requirement "The base (of the desk spot lamp) should provide support to the artifact":

   $req\_exp(R) \equiv (\forall\ p)\ base(p) \supset (\exists\ f)\ provide\_support\_feature(f) \wedge feature\_of(f, p)$.

5. *Cost requirements* give restriction on the cost of manufacturing the artifact. There may be requirements on cost of designing and assembling as well. Cost requirements are sometimes important factors in design. They are reflected in choice of material (economical vs. expensive) and introduction of features (simple vs. sophisticated).

   "The total cost of manufacturing the artifact (desk spot lamp) should be no more than $50":

   $req\_exp(R) \equiv (\forall\ p)\ desk\_spot\_lamp(p) \supset cost(p) \leq 50.$

5.5 Constraints

Constraints are statements that must be satisfied by design. Since it puts restriction on the design, each primitive requirement is also viewed as a constraint. That is, requirements are decomposed into constraints at the final step of the requirement decomposition process. In addition to the constraints that decompose from requirements, there are constraints that capture various physical laws that must always be obeyed by the design. For example, if the artifact is a geometrical object, it has to satisfy laws of geometry and topology, often described as equations or inequalities over parameters of the artifact. The physical laws can also be used to derive knowledge that is previously unknown to us, e.g. if two angles in a triangle structure are given then the third angle can be calculated by invoking the triangle principle. These constraints are thus also called *deductive rules*. The content of each constraint is described by a logical sentence. We call it *constraint expression* and denote by *con_exp(c)* for a constraint *c*. Constraint expressions are first order logic sentences. We do not have restriction on the formats of the sentences. Since requirements have been discussed earlier, in the next (sub)-section we list several examples of constraints that are not (primitive) requirements.

5.5.1 Several Examples of Constraints

One interesting group of constraints are those related to, and can be inferred from, the structure of parts. The simplest one is that the weight of a part is equal to the sum of the weights of its (direct sub-)components. Suppose $p_1,..., p_n$ are direct components of part p. Then:

$$weight(p) = \sum weight(p_i). \qquad (9)$$

The cost of a part is equal to the sum of the costs of its (direct) components and the cost of assembling the components into the part: [2]

$$cost(p) = \sum cost(p_i) + assembly\_cost(p).$$

We can also calculate the power consumption of a part from that of its components. The relation, though, may not be as simple as that of weight or cost. In addition, there may be several levels of power consumption, depends on the modes of the components are on, e.g.

2. There are many models of cost calculations, some of which are fairly complex, involving labor rate, recurring costs, etc. Here we only demonstrate an idea of how to specify a cost model in terms of axioms.

operating mode, standby mode and resting mode, which we would not discuss the detail here. In summary, each constraint in this group captures a relationship between $\alpha(p)$ and $\alpha(p_1),..., a(p_n)$, where $p_1,..., p_n$ are direct components of part p and α is a property of the parts. The relationship is generally a function f such that

$$\alpha(p) = f(\alpha(p_1),..., \alpha(p_n)).$$

Another group of constraints are *domain constraint* for parameters. Each parameter is associated with one of the domain constraints. For example, the following is a constraint on the parameter "weight", which says weight of a part must be positive.

$$(\forall\ p)\ weight(p) > 0. \tag{10}$$

"The base cover of the lamp must be built of the material either cast iron or cast steel":

$$materal(Base\_cover) = Cast\_iron \vee materal(Base\_cover) = Cast\_steel. \tag{11}$$

"The color of *Heavy_base* must be one of {blue, white, black}":

$$color(Heavy\_base) = Blue \vee color(Heavy\_base) = White \vee color(Heavy\_base) = Black. \tag{12}$$

5.5.2 Relationship of Parametric Constraints and Parts

Parametric constraint is a special class of constraint; it is largely concerned with *parametric design*, where an artifact is characterized by a set of parameters and a set of constraints that limit the values of these parameters [7]. As defined, parametric constraints are constraints whose expressions have no variables [7]. According to this definition, the constraints (11) and (12), and that from the primitive requirement (2) are parametric constraints, while (10) and (8) are not. In talking about the relationship of constraints and parts, we restrict ourselves on parametric constraints, since for more general constraints the relationship is difficult to discuss due to the arbitrary form of the constraint expressions.

The relationship of constraints and parts will be brought out by the notion of *domain* of parametric constraint. The domain of a (parametric) constraint is in a sense similar to the domain of a parameter (which is the set of values that can be achieved by the parameter). The domain can be roughly thought of as the set of objects (parameters with their parts or features) that the constraint is concerned with. Since a constraint puts restrictions on certain parameters, the domain can also be viewed as the set of parameters (with the parts or features that the parameters belong to) that the constraint has restrictions on.

Let *domain(cr)* denote the domain of *cr*, where *cr* a constraint or a requirement.

We first define the domain of a parametric constraint. The domain of a constraint *c* is defined as the set of objects of the form *para(pf)* that appear in *con_exp(c),* where *para* is a parameter name and *pf* is a part or a feature. Likewise, for a primitive requirement *r* that is a parametric constraint, the domain is defined as the set of objects of the form *para(pf)* that appear in *req_exp(r)*.

For instance, the constraint (11) has the domain *{materal(Base_cover)}*, (12) has the domain *{color(Heavy_base)}*, and (2) has the domain *{weight(Desk_spot_lamp)}*.

The domain of a non-primitive requirement is the union of the domains of its decompositions. Suppose $r_1, \ldots, r_n$ are the direct decompositions of r. Then:

$$domain(r) = domain(r_1) \cup \ldots \cup domain(r_n).$$

Note that this definition of domain is a syntactical one. It might be the case that a parameter is in the domain of a constraint (or a requirement) but the parameter is not restricted by the constraint (or the requirement). For example, suppose we have written the following constraint *C* with the expression:

$$color(Heavy\_base) = Blue \ \vee color(Heavy\_base) \neq Blue.$$

Although $color(Heavy\_base) \in domain(C)$, it is easily seen that the constraint does not have any effect on the color of Heavy base. This kind of constraints are tautologies; they have no meaning and should be avoided to write.

With the domain definition, the ontology can answer the following question: Does a requirement R impose a constraint on part P? Assuming that the requirement is decomposed into parametric constraints, this can be answered by finding out whether there is a parameter name *para* such that:

$$[para(P) \in domain(R)] \vee (\exists f)\, feature\_of(f, P) \wedge [para(f) \in domain(R)]. \qquad (13)$$

6.0 Example Queries

We have developed a terminology for requirements and specified axioms among the terminology. We also present the terminology and axioms for parts, features, and constraints. This forms an ontology centered on requirements in engineering design. The ontology can be used for answering many common sense questions, by deduction using a theorem prover. In this section we demonstrate the queries that we listed in 3.1. We will emphasize the use of axioms in answering these queries. The ontology is implemented in Prolog in an object-oriented fashion similar to ROCK knowledge base system from Carnegie Group. In this implementation, predicates and functions are expressed in some uniform format which may not be as those appear above. For example, `attribute(r1, req_exp,'weight(desk_spot_lamp)<2')` is the implementation of $req\_exp(r1) \equiv weight(desk\_spot\_lamp)<2$, `attribute(clip1, weight,0.02)`the implementation of $weight(clip1) = 0.02$, and `relatioin(r11, has_decomposition,[r21,r22])` the implementation of *decomposition_of(r21, r11)* and *decomposition_of(r22, r11)*. We keep this format to simulate the actual output of the system.

- **Requirement refinement:**

  Question 1:

  Input: `attribute(r1,req_exp,'weight(desk_spot_lamp)<2')`

`attribute(r11,req_exp,'weight(heavy_base)<=1.3')`
`attribute(r12,req_exp,'weight(short_arm)<0.3')`
`attribute(r13,req_exp,'weight(samll_head)<=0.4').`

Query(E): Do the decomposed requirements `r11,r12,r13` preserve the meaning of requirement `r1`?

Query(P):`?- faithful_decomposition(r1,[r11,r12,r13]).`

Output: `not.`

Axioms: In order for a decomposition to be faithful, it must satisfy both axioms (3) and (4). The decomposition satisfies (3) but not (4), and hence does not preserve the meaning of the original requirement. ∎

Question 2:

Input: `attribute(r1,req_exp,'weight(desk_spot_lamp)<2')`
`attribute(r2,req_exp,'weight(desk_spot_lamp)<2.2')`

Query(E): Is there a redundant requirement between `r1` and `r2`?

Query(P):`?- subsume(r1,r2);subsume(r2,r1).`

Output: `yes.`

Axioms: Since `subsume(r1,r2)` is true by axiom (7), `r2` is a redundant requirement. ∎

Question 3:

Input: `customer(a12),req_posted_by(r1,a12).`

Query(E): Is `r1` an explicit statement of the customer?

Query(P):`?- explicit(r1).`

Output: `yes.`

Axioms: From `customer(a12)` and `req_posted_by(r1,a12)`, it is concluded `external(r1)` by axiom (5), and therefore `explicit(r1)` holds by axiom (6). ∎

- **Requirement traceability:**

Question 1:

Input: `relation(r1,has_decomposition,[r11,r12,r13])`
`relation(r11,has_decomposition,[r21,r22])`
`relation(r12,has_decomposition,[r23,r24])`

Query(E): Does requirement `r1` decompose to others?

Query(P):`?- decomposition_of(X,r1).`

Output: `X=[r11,r12,r13,r21,r22,r23,r24].`

Axioms: The transitivity axiom of *decomposition_of* results in that `r21,r22,r23,r24` are also decomposed requirements of `r1`. ∎

Question 2:

Input: as in Question 1.

Query(E): What is the source of `r24`?

Query(P): `?- req_source(r24,X).`

Output: `X=r1.`

Axioms: It is concluded from the transitivity axiom of *decomposition_of* that `r24` is a decomposed requirement of `r1`. From the definition axiom of *req_source*, it is then concluded that `r1` is the source of `r24`. ▮

Question 3 & 4: Simple queries involving axioms of *req_posted_by(r, a)*, *external(r)* and *internal(r)*. Omitted. ▮

- **Requirement satisfaction:**

  Question 1 is a special case of Question 2.

  Question 2: Generally, for the question "Are two requirements R_1 and R_2 in conflict?", the system first gets the *primitive decompositions* of R_1 and R_2 (i.e. the primitive requirements that are the decompositions of R_1 and R_2). Suppose R_{11},..., R_{1n} and R_{21},..., R_{2n} are the primitive decompositions of R_1 and R_2 respectively. Then the query becomes the following problem:

  Is $req\_exp(R_{11}) \wedge ... \wedge req\_exp(R_{1n}) \wedge req\_exp(R_{21}) \wedge ... \wedge req\_exp(R_{2n})$ consistent?

  This reduces to the classical problem of checking consistency of a first-order sentence.

  Input: `attribute(r1,req_exp,'weight(desk_spot_lamp)<2')`

  `attribute(r4,req_exp,'weight(desk_spot_lamp)>3')`

  Query(E): Are requirements r1 and r4 in conflict?

  Query(P): `?- conflict_requirements([r1,r4]).`

  Output: `yes.`

  Axioms: The primitive decompositions of r1 and r4 are themselves. Since $req\_exp(r1) \wedge req\_exp(r4)$ is not consistent, the two are declared conflicting. ▮

  Question 3: Suppose R is a requirement and R_1,...,R_n are primitive decompositions of R. Then R is satisfied iff the sentence $req\_exp(R_1) \wedge ... \wedge req\_exp(R_n)$ is true and R is violated iff the sentence is false. Example omitted. ▮

- **Version & Change:**

  Question 1 & 2: The system follows the *derived_from* link to retrieve the older versions and use *creation_time* predicate to determine the latest version. Example omitted. ▮

  Question 3: The system checks if the new set of requirements is still satisfiable. Example omitted. ▮

- **Relationships of requirements to parts:**

  These queries can be answered in parametric design. The first query has been discussed in Section 5.5.2. The second query are of the same nature as the first one except that it is necessary to find out all the requirements that impose a constraint on the part. The answer to the third query is the parameter name *para* in (13).

- **Product structure:**

  Question 1 & 2: The system uses the axioms for predicates *component_of* and *feature_of*.

Question 3 & 4: Similar to the questions in relationships of parts and requirements.

Question 5:

Input: `relation(desk_spot_lamp, has_component,[heavy_base, short_arm, small_head]),`

```
        relation(heavy_base,has_component,[clip1,base_cover, weight_-
  disc])
        instanceOf(clip1,clip)
```

Query(E): Where is the part with the type clip used?

Query(P): `?- component_of(P,X),instanceOf(P,clip).`

Output: `X=heavy_base; desk_spot_lamp.`

Axioms: The question is to find out those parts that have a component of the type *clip*. The transitivity axiom of the *component_of* relation allows the system to traverse the links in the component tree. ∎

Question 6:

Input: `relation(desk_spot_lamp, has_component,[heavy_base, short_arm, small_head]),`

```
        relation(heavy_base,has_component,[clip1,base_cover, weight_-
  disc])
        attribute(clip1,weight,0.02)
        attribute(base_cover,weight,0.1)
        attribute(weight_disc,weight,0.7)
        attribute(short_arm,weight,0.3)
        attribute(small_head,weight,0.2)
```

Query(E): What are components of desk_spot_lamp that exceed 20% of the total mass?

Query(P): `?- component_of(X,desk_spot_lamp),weight(X)> 0.2*weight(-desk_spot_lamp).`

Output: `X=heavy_base; short_arm; weight_disc.`

Axioms: The weights of heavy_base and desk_spot_lamp are computed from the axiom (9). ∎

The axioms in our ontology also allow integrity checking of the design data. Some of the data provided by the engineers may be invalid. Furthermore, update to the knowledge base may introduce inconsistency among the data and the object model for design. For example, suppose the user tries to provide a data such as *decomposition_of(R143, R143)*. This violates the axiom stating that the *decomposition_of* relation is non-reflexive, and therefore can be detected easily.

7.0 Conclusion

In order to make design knowledge effectively accessible across an enterprise, the knowledge needs to be classified, defined and related in a well-defined terminology acceptable by all participating engineers. In this paper we have described an ontology for requirements in the engineering design domain. We use first-order logic to define components of the ontology, and identify the axioms involved in the objects and their interactions with the aim of answering common sense questions.

We discuss the issues raised by a generic requirements management process and how our requirement ontology addresses these issues. The ontology provides communication of requirements by defining a well-defined syntax and semantics. It addresses traceability issues by providing explicit relations for it and allows for checking for satisfiability or consistency. It provides a knowledge-base for tools that perform document creation and tools that are responsible for managing change.

8.0 References

[1] Bilgic, T and Fox, M. S. Constraint-based retrieval of engineering design cases: context as constraints (1996) to appear in J. Gero and F. Sudweeks (eds) *Artificial Intelligence in Design '96*, Kluwer Academic Publishers. http://www.ie.utoronto.ca/EIL/public/ aid96/cbret1.html

[2] Dixon J.R., Cunningham J.J., Simmons M.K., Research in designing with features, in Intelligent CAD I, eds. Yoshikawa H., Gossard D., *Proc. IFIP TC 5/ WG 5.2 workshop on intelligent CAD*, Elsevier, 1987, 137-148.

[3] Fiksel J., Hayes-Roth, F., Computer-aided requirements management, *Concurrent Engineering: Research and Applications* (1993), 1:83-92.

[4] Fox, M., Chionglo, J.F., and Fadel, F.G. "A Common Sense Model of the Enterprise", *Proceedings of the 2nd Industrial Engineering Research Conference,* pp. 425-429, Norcross GA: Institute for Industrial Engineers. http://www.ie.utoronto.ca/EIL/papers/abstracts/14.html

[5] Fox, M.S., Finger, S., Gardner, E., Navin chandra, D., Safier, S.A., and Shaw, M., "Design Fusion: An Architecture for Concurrent Design", in *Knowledge-aided Design,* Academic Press Ltd., London, UK, edited by Green, M., pp. 157-195, 1992.

[6] Fox, M.S., Salustri, F.A. "A One-Off Systems Engineering Model", *AAAI Workshop on Artificial Intelligence and Systems Engineering*, August 1994, Seattle, Washington. http://www.ie.utoronto.ca/EIL/papers/abstracts/33.html

[7] Gruber, T. R. and Olsen, G. R. The configuration design ontologies and the VT elevator domain theory. *International Journal of Human-Computer Studies* 44, 569-598, 1996.

[8] Gruninger, M., and Fox, M.S., (1994), "The Design and Evaluation of Ontologies for Enterprise Engineering", *Workshop on Implemented Ontologies, European Conference on Artificial Intelligence (ECAI) 1994*, Amsterdam, NL. http://www.ie.utoronto.ca/EIL/public/onto_ecai94.ps

[9] Gupta, L., J. Chionglo, and M. S. Fox (1996) A Constraint Based Model of Coordination in Concurrent Design Projects, to appear in the *Proceedings of WET-ICE'96*. http://

www.ie.utoronto.ca/EIL/DITL/WET-ICE96/ProjectCoordination/WETICE96_Project-Coordination.fm.html

[10]Gwizdka, J., L. Louie, and M. S. Fox (1996), EEN: A Pen-based Electronic Notebook for Unintrusive Acquisition of Engineering Design Knowledge, to appear in the *Proceedings of WET-ICE'96*. http://www.ie.utoronto.ca/EIL/DITL/WET-ICE96/ EEN/ EEN_WetIce96.html

[11]Hoffman, D. A overview of concurrent engineering. *Tutorial Notes of 1994 Annual Reliability and Maintainability Symposium*, California, January 1994.

[12]Kott, A. and Peasant, J. L. Representation and management of requirements: The RAPID-WS project. *Concurrent Engineering: Research and Applications*. Vol. No. 2. Pages 93-106. June 1995.

[13]Product Data Representation and Exchange- Part 44 - Integrated Resources: Product Structure Configuration, ISO 10303-44, 1992.

[14]Salomons O.W., Houten F.J.A.M. van, Kals H.J.J., Review of research in feature-based design, *Journal of Manufacturing Systems*, Vol.12, No. 2, 1993, 113-132.

[15]Wieringa, R. J., *Requirements Engineering: Frameworks for Understanding*, John Wiley and Sons, New York (1996).

[16]Yen, J., Liu, X. and Teh, S. H. A fuzzy logic-based methodology for the acquisition and analysis of imprecise requirements. *Concurrent Engineering: Research and Applications* (1994) 2, 265-277. Wiley and Sons, New York.