A Product Ontology

Jinxin Lin, Mark S. Fox and Taner Bilgic
Enterprise Integration Laboratory, Dept. of Mechanical & 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

Draft: 7 July 1997

1.0 Competency of the Ontology

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

1. **Product structure:**

   What are the components of a part? What features do the part has? What constraints that the part must satisfy? Where a specific type (or class) of parts are used? What are components of Assembly P that exceed a certain percentage of the total mass?

2. **Parameter:**

   What is the domain of a parameter? What constraints that parameter-X of the part must satisfy? What is the units of the parameter? What is the value of the parameter in unit X? What is the type of the parameter? Is an equation dimensionally homegenous?

3. **Requirements and constraints:**

   Who posted the requirement, and when it was posted? Is the requirement implicit or explicit requirement, hard or soft requirement? Is its scope internal or external? What is the method of verifying the requirement? What is its current status?

   Is the constraint satisfied or violated? What is the strength of the constraint? Who posted the constraint and when it was posted? Is it currently active in the constraint network? From which requirement the constraint is decomposed?

4. **Relationships of requirements and constraints to parts:**

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

5. **Functions:**

   What does this part do? What is this component for? Can we use another part instead of this one that does the same job?

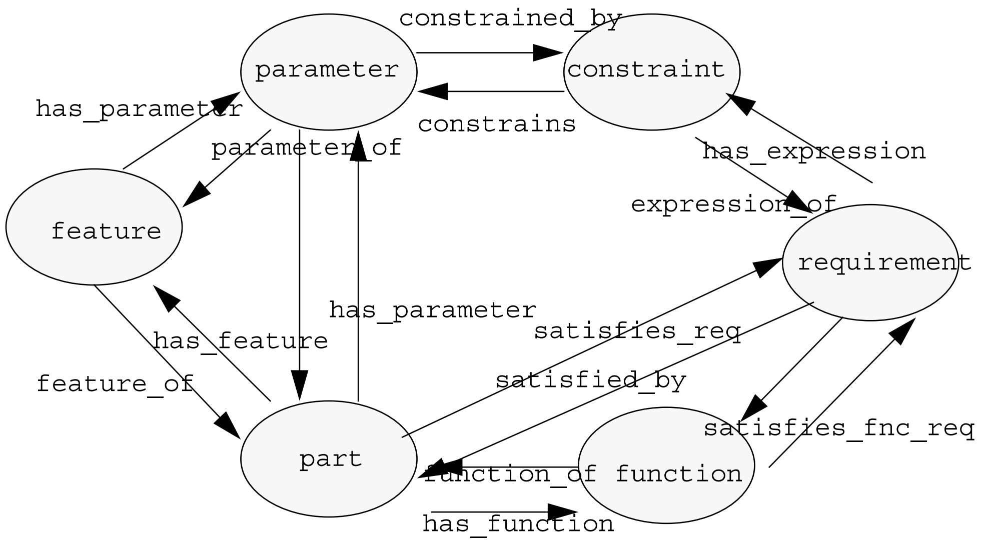
6. **Version & Change:**

   Is this the latest version of a part? Why have they changed? Are two versions parallel versions? Does a change affect consistency of the requirements set?

Objects included in our ontology are *parts, features, parameters, requirements* and *constraints*. Each of which can also have *versions* describing the evolution of the objects. Parts are components of the artifact or the artifact itself. Features are used to describe geometrical, functional and other characteristics associated with a part. Both part and feature can have parameters that define their properties such as weight, color, diameter, material, surface finish, etc. Requirements specify the properties (functional, structural, physical, etc.) of the artifact being designed. Constraints form the leaves of the requirement decomposition tree, and embody physical laws, equations etc.

Parts, features, parameters, requirements and constraints are related to each other with corresponding relationships, as depicted in Figure 1.

FIGURE 1. Relationship of the product object representation



2.0 Product Structure

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, parameter, 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 *spar_part* is a subclass of the class part: *subclassOf(spar_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 *spar_part*: *instanceOf(PRT131, spar_part)*.

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

2.1 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.

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* and the *composed-of* relations defined in [9] and [16] 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*:

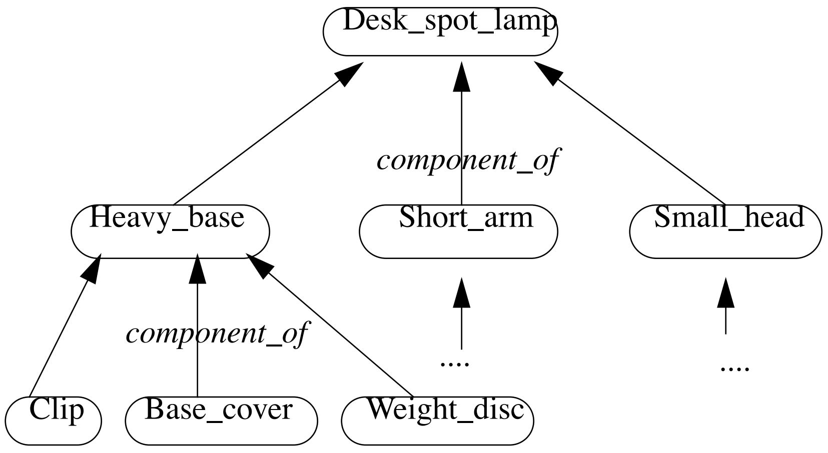
component_of(Heavy_base, Desk_spot_lamp).

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:

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

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*.

$$(\forall\ p_1, p_2)\ has\_component(p_1, p_2) \equiv component\_of(p_2, p_1). \tag{1}$$

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 make 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.

A part, either primitive or composite, is associated with the following information:

- part_description: descriptive information of the part other than its ID;
- part_creation_date: the date of the first creation;
- part_designer: the original designer of the part;
- modification_date: the date of last modification;
- last_modified_by: the agent (a person or a team) who last modified the part;
- part_owner: the agent who currently has control of the part;
- version_number: the version number of the part;
- design_document: document presenting design process, rationale, etc. Usually a file name;
- image_name: symbology of the part; usually the name of a file (in the format e.g. gif, bmp, etc.)

The following lists some axioms with respect to the above information of part:

$(\forall\, p)\ (\exists\, a)\ agent(a) \wedge a = part\_owner(p).$

"There must be at least one agent who is the owner of a part."

$(\forall\, p, p')\ component\_of(p', p) \supset part\_creation\_date(p') < part\_creation\_date(p).$

"Every component must be created earlier than its assembly."

$(\forall\, p, p')\ component\_of(p', p) \supset modification\_date(p') < modification\_date(p).$

"If a component of a part is modified then the part is modified."

A part can have many *parameters* presenting the physical nature of the part; this we describe in one of later sections.

2.2 Features

There are different kinds of features associated with a part, e.g., geometrical features, functional features, assembly features, mating features, physical features, etc. [3] [18]. 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 a part; they define what the part can be used for.

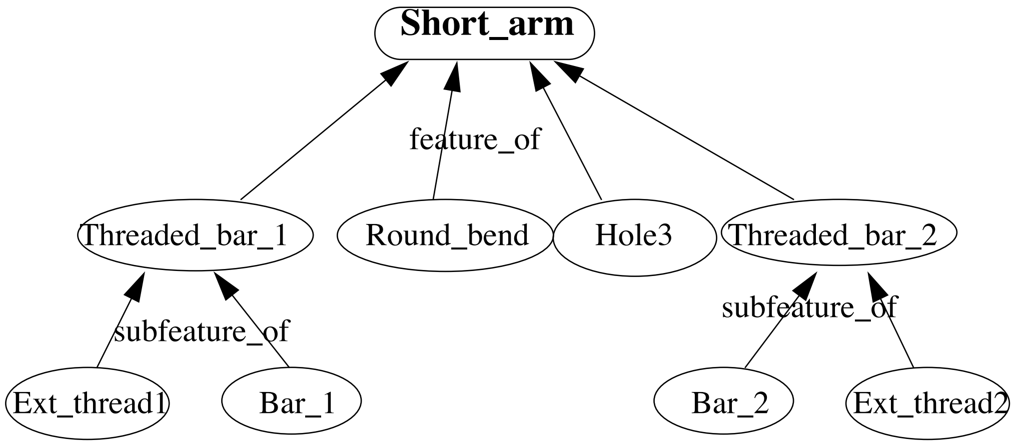
A part and its features are related by the predicate *feature_of(f, p)*, saying that *f* is a feature of part *p*. The following term represents the fact that a hole feature, called *Hole3*, is introduced in the short arm of the desk spot lamp:

feature_of(Hole3, Short_arm).

There can be compound 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 gives the definition of compound features and atomic features:

$(\forall f)\ compound\_feature(f) \equiv (\exists f_1, f_2)\ subfeature\_of(f_1, f) \wedge subfeature\_of(f_2, f) \wedge f_1 \neq f_2.$
$(\forall f)\ atomic(f) \equiv \neg\ (\exists f')\ subfeature\_of(f', f).$

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.

A feature is associated with the following information:

- feature_description: descriptive information of the feature;
- feature_introduce_date: the date of associating the feature with the part;
- feature_type: the type of a feature, e.g. geometrical, topological features, etc.
- feature_designer: the original designer of the feature;
- feature_owner: the agent who currently has control of the feature;
- design_document: document with full detailed information of the feature;
- image_name: symbology of the feature; usually the name of a file (in the format e.g. gif, bmp, etc.)

2.3 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)*.

The parameters such as color, money, etc. are not a *physical quantity*. *Physical quantity*, as stated in [15], are something that can be measured by some strictly definable process. In real world, a majority of parameters are some kinds of physical quantitiy such as mass, time, length, volume, density, force, energy, speed, acceleration, temperature, electric current, etc. Any physical quantity has two essential components: the *magnitude* of the quantity and the *unit of measurement*. Magnitude, which we also call *value*, is not meaniful by itself for the quantity without mentioning the unit.

A parameter is associated with the following information:

- parameter_description: descriptive information of the parameter;
- associated_object: the part or feature that the parameter is associated;
- unit: the unit of measurement, if any, of the parameter, e.g. pound, kg, liter, etc.;
- value: the actual value of the parameter;
- type of value: the type of the value of the parameter, e.g. nominal, ordinal, interval, rational, or boolean (more on this later);
- parameter_owner: the agent who controls the parameter, i.e. the agent who has the authority to change, delete the parameter;
- design_document: document with full detailed information of the parameter;
- image_name: symbology of the parameter.

The information is recorded by the predicates:

property(para_name, slot_name, slot_value),

where *slot_name* is one of the catogery of above information, and *slot_value* is the recorded value of that catogery. For example, we have a parameter named *WP11* which is the diameter for the feature Hole3:

property(WP11, associated_object, Hole3),

property(WP11, unit, Centimeter),

property(WP11, value, 3),

property(WP11, parameter_owner, Design_team_4),

...

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.

2.3.1 Units

A system of units consists of a few fundamental units, called *base* units, from which other unites are derived (called *derived* units). In the System International (SI), some base units are length (m), mass (kg), time (s), electric current (A), temperature (K), etc.

si_base_unit(length, metre),

si_base_unit(mass, kilogram),

si_base_unit(time, second),

si_base_unit(electric_current, ampere),

si_base_unit(temperature, kelvin),

...

Other units are defined in terms of the base units or other derived units, for example, the unit of velocity is the unit of length divided by the unit of time which is m/s. The unit of acceleration is the unit of velocity divided by the unit of time again (i.e. m/s^2), and the unit of force is the unit of mass times unit of accelaration (i.e. kg m/s^2, called “Newton” denoted by N).

si_derived_unit(force, N),

si_derived_unit(stress, Pa), (Pa = N/m^2*)*

si_derived_unit(energy, J), (J = Nm)

...

A SI unt is either a SI base unit or SI derived unit:

$(\forall\ d, u)\ si\_unit(d, u) \supset si\_base\_unit(d, u) \vee si\_derived\_unit(d, u).$

Units of the same kind in different systems can be converted to each other through the so-called *conversion factors*. A conversion factor is associated with two units and one numeric value, and stored in the predicate *conversion_factor(u1, u2, factor)*. For example, we have *conversion_factor(yard, metre, 0.9144),*

conversion_factor(pound, kilogram, 0.45359237).

...

(Where the first means that one yard is equal to 0.9144 metre, and the second means that one pound is equal to 0.45359 kilogram.)

Note that the conversion factor of the same unit is always 1:

$(\forall u)$ *conversion_factor(u, u, 1)* .

With the conversion factors, we can obtain the value of a parameter in any unit:

$(\forall p, u, v)$ *value(p, u, v)* $\equiv (\exists u', v', factor)$ *property(p, unit, u')* $\wedge$ *property(p, value, v')* $\wedge$ *conversion_factor(u, u', factor)* $\wedge\ v = factor * v'$.

where *value(p, u, v)* means that the value of a parameter *p* is *v* in unit *u*.

2.3.2 Dimensions

As mentioned, a phsical quantity is characterized by both its *value* (magnitude) and its *unit*. In fact, unit plays a more important role in distinguishing one quantity from another. For example, “12 inch” denotes a measurement of *length*, not a measurement of *mass* (we understand this is from the unit “inch”, not from the value “12”). Note that there may be many possible units for one quantity, e.g. a unit for length may be “inch”, “metre”, “mile”, “foot”, etc. Massey [15] introduces the notion of *dimention* of a quantity to mean a unit of the quantity without mentioning which particular unit it is. He uses symbol ‘[X]’ to denote the dimention of a physical quantity X. The dimension of a physical quantity is usually defined in terms of the dimensions of other physical quantities. For example, velocity of a moving object is calculated as follows

$v = l / t$

where l is the length traversed by the object and t is the time spent. Hence the dimension of velocity is $[L][T]^{-1}$, where [L] represents the dimension of length and [T] represents the dimension of time. And therefore every physical quantity has a dimension. Two quantities are comparable if they have the same dimension:

comparable_quantities(q1, q2) $\equiv [q1] = [q2]$.

In addtion, given any formula α, its dimension can be calculated. We also denote the dimension of α by $[\alpha]$.

Frequently, engineers use mathematical *equations* to describe the relationships among physical quantities and to express physical laws or associations. There is, therefore, a problem of *dimensional homogeneity* of an equation. As defined in [15], an equation is *dimensionally homogeneous* if any terms in the equation that are added, subtracted or equated must represent magnitudes that may be expressed in terms of the same unit. For example, it makes no sense to add a length and a mass, or a force and a velocity. In the following we will capture the notion of *dimensional homogeneity* in terms of axioms in our ontology. For that, we need to define the *validity* of a formula (an equation is also treated as a formula within which there is an equity operator "="). We use the predicate *valid*(α) to means that the formula α is valid. Then for a formula in the form of "$\alpha 1+\alpha 1$", "$\alpha 1- \alpha 1$", *or* "$\alpha 1=\alpha 1$", we have:

$$valid(``\alpha 1+\alpha 1") \equiv valid(\alpha 1) \wedge valid(\alpha 2) \wedge [\alpha 1] = [\alpha 2].$$

$$valid(``\alpha 1-\alpha 1") \equiv valid(\alpha 1) \wedge valid(\alpha 2) \wedge [\alpha 1] = [\alpha 2].$$

$$valid(``\alpha 1=\alpha 1") \equiv valid(\alpha 1) \wedge valid(\alpha 2) \wedge [\alpha 1] = [\alpha 2].$$

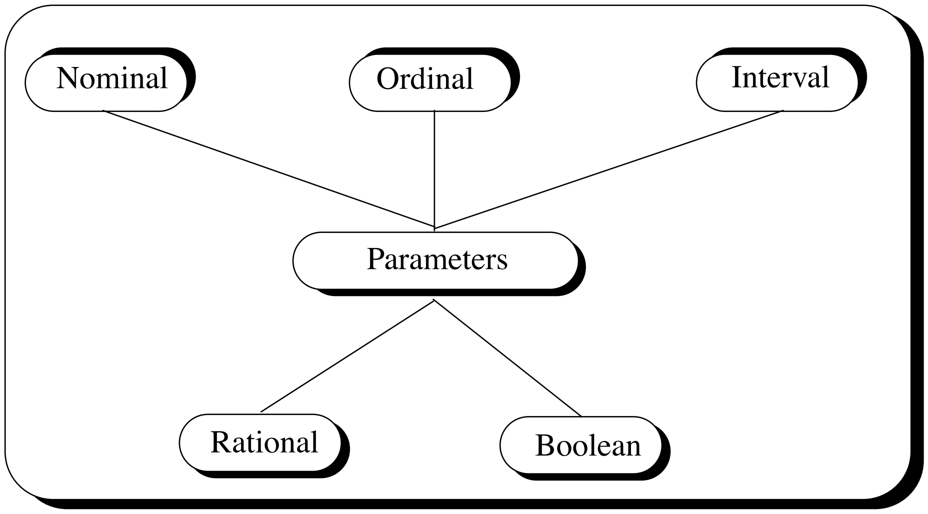
Definition of Dimensional Homogeneity: An equation α is dimensionally homogeneous if *valid*(α) **is true:**

$$dimensional\_homogeneous(\alpha) \equiv valid(\alpha).$$

2.3.3 Classification of Parameters

In representing parameters, five different types can be distinguished, namely, nominal, ordinal, interval, rational, and boolean parameters.

FIGURE 4. Classification of Parameters



- Nominal Parameters: In nominal scales, numbers are used just as strings, like names of things. For example, in manufacturing, a machine may be assigned a label 162314. The number of the label is merely a substitute for the name of the machine, and it has no other meaning[1]. The only operation we can do with these numbers is to count them, we cannot add, subtract or multiply them with any other number.

1. The coding schemes are exception of this.

- **Ordinal Parameters**: In ordinal scales, only the ordinal properties of the numbers are employed in measurement. Numbers have meanings according to the ordering of the property being represented. For example, the ease of use for a machine can be represented on an ordinal scale. On a scale of 1-7, 1 may represent “most difficult to use”, and 7 may represent “easiest to use”. By looking at two numbers on this scale, we can tell which machine is easy to use relatively.
- **Rational Parameters**: In ratio scales, scale values can be uniquely determined except for an arbitrary unit of measurement. Such scales are very common; for example, once we determine the unit of measurement for the length of a robot arm as centimeters, the value of the length parameter can be uniquely determined. Ratio scales not only have equality of units, but also have an absolute zero for that unit of measurement. For example, Time has an absolute zero, and units of time are equal to each other.
- **Interval Parameters:** Interval parameters have their domains as an interval, and the possible value that it can take could be a subinterval, or an exact value within that interval. An interval is a set [a, b] such that all the real numbers between a and b are included in the set. Intervals can be operated by interval arithmetic operators and set theoretic operators. An interval of a function provides the upper and lower bounds for the range of the function, when its arguments span an interval.
- **Boolean parameters**: Boolean parameters take values as TRUE or FALSE. Propositional logic forms the basis of evaluation and analysis of boolean parameters and expressions. For example, a parameter requirement_satisfied may be defined for each of the design requirements for the lamp, which takes a value True, if the requirement is satisfied, and False otherwise.

3.0 Functions

Webster dictionary define the word function as: The *action* for which a component is particularly fitted or employed. In accord with that definition, most of the engineering design literature take function to mean the *intended behaviour* of the artifact.

A function is usually defined by a verb and a noun (e.g. increase pressure, transfer torque). In general, function of an artifact can be defined using activities, effects, goals, and constraints.

In the most sophisticated case, the function representation gives way to a detailed *simulation* of the artifact. However, in complex systems this level of detailed representation may be unattainable or simply undesired. It is important to put forth *reasons* to represent function for a particular application.

Determining a functional representation is usually possible using a top-down approach. First, the overall task of the artifact is identified and then the overall function corresponding to the overall task is decomposed into sub-functions (Pahl&Beitz, 1988). The resulting structure is called the *function structure*. The method of this decomposition is not clear and tends to be different in one-off designs and adaptive designs. For one-off designs the basis of a function structure is the *specification* and the *abstract formulation* of the problem. In adaptive design, however, the existing function structure yields a lot of information to be interpreted and adapted. In the former case, the main use of functional structures is simple *classification*.

The level of abstraction in representing functions is important in reusing older design solutions for new problems. The higher level functions should be independent of the domain as much as possible. Domain specific functions should be introduced as higher level functions decompose to their sub-functions. For example, a higher level goal can be as general as "deploy payload".

3.1 Why do we need to represent functions?

There are various reasons for each application to represent functions and this necessitates representations at different levels of abstraction. The following are some reasons to represent functions:

- classification
- finding a design solution
- design validation

In engineering design, sometimes there is a need to *store* artifacts with respect to what they do. In such a case, the function representation is part of the product representation. It is another view of the product information.

Design search space is usually traversed by function. The designer has a function in mind and the conceptual stage of the design is dominated by searching for the right concept which provides the right functionality.

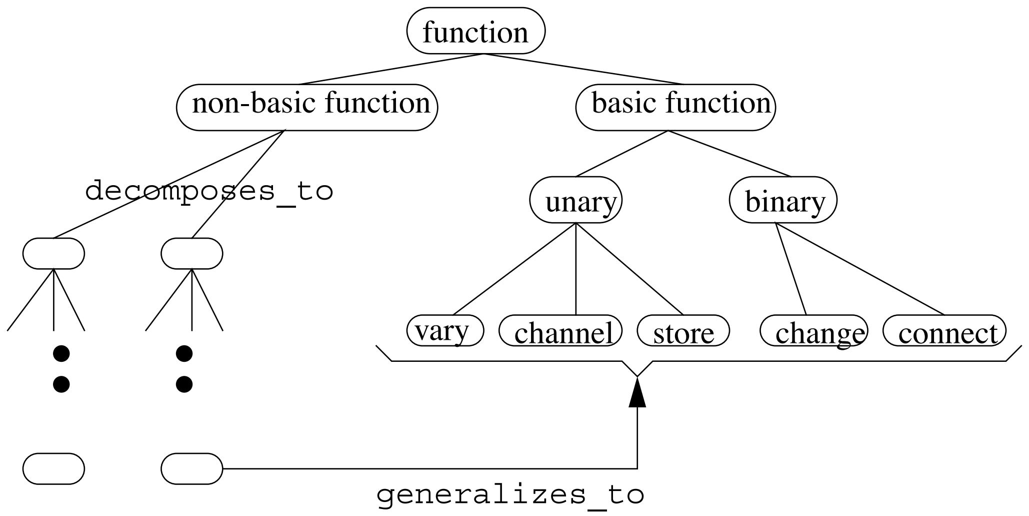
When the aim is to validate a design concept one can take a detailed simulation approach or simply use qualitative simulation at the generic functional level.

In TOVE, we employ the *classification* view of functions with the goal of viewing design artifacts with respect to their functionality. For this purpose, we use an input/output representation of functions.

3.2 Function as input/output

German researchers have been very active in design methodology research and during the course of their investigation they came up with a conceptualization of function as an input/output process. Pahl and Beitz (1988) summarizes the findings of German researchers. Our representation is based on this conceptualization with a few extensions.

FIGURE 5. Function representation in TOVE



The aim of the function representation reported in (Pahl and Beitz, 1988) is to aid the novice designer in the design process. The design process is envisioned as guided by the functions and their decompositions (the so called function structures).

Function structures are obtained when one decomposes a high level function to its sub-functions. In turn, the sub-functions are related to each other in various ways. A functional structure shows *how* a function is accomplished rather than a process flow. There are two main processes when coming up with a function structure: (i) function decomposition, (ii) function allocation. In TOVE, the notion of a non-basic function serves for this purpose. Non-basic functions are totally user definable, indexed by their names and synonyms. Related to each other via `decomposes_to` relation to denote the function structure.

However, each non-basic function should (or have a parent that does) generalize to a basic function. Basic functions are distinguished, "generally valid" functions as identified in Pahl and Beitz (1988):

- **change**: The characteristic of this function is the *type*. The type and the outward form of input and output differ.
- **vary**: The characteristic of vary is *magnitude*. This function is relevant when considering energy and signals.
- **connect**: The characteristic of this function is the *number*. It connects two or more inputs.
- **channel**: The characteristic of channel is *place*. The place of the input is no more the same once this function is performed.

- **store**: The characteristic function of store is the *time*. Energy, material or signals are stored for a period of time.

TABLE 1. Basic functions

Basic function	Argument(s)
Vary, Channel, Store	Energy Matter Signal
Change	Energy-energy Matter-matter Signal-signal
Connect	Energy-signal Matter-energy Matter-signal Matter-matter Signal-energy Signal-matter Signal-signal

4.0 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.

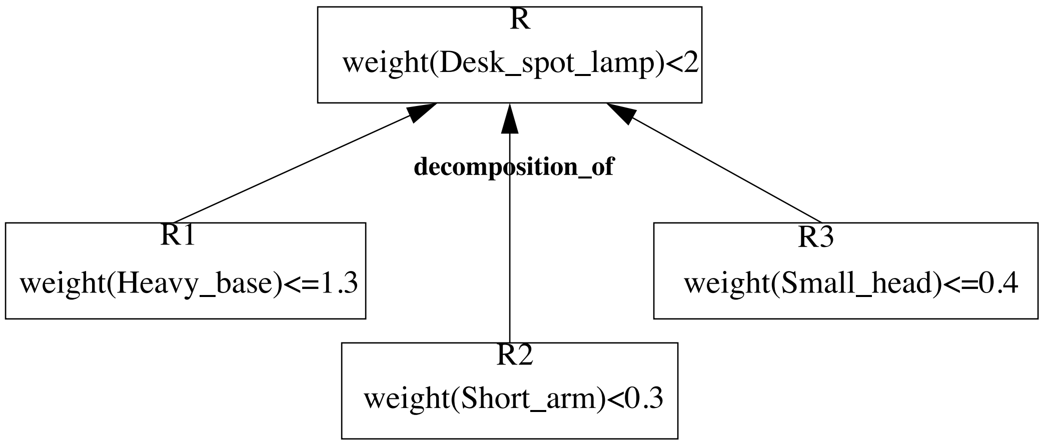
4.1 Decomposition of Requirements

The hierarchy of requirements is built on the relation *decomposition_of*. Figure 6 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.)

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 6. Weight Requirement Decomposition



* Unit of measurement: pound

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".

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

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 6 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.

4.2 Requirement information

A requirement is associated with the following information:

- req_description: descriptive information of the requirement;
- req_doc: the document associated with the requirement containing textual and graphical description of a requirement;
- req_posted_by: the agent who posted the requirement;
- req_posting_time: the date or time when the requirement is posted;
- req_scope: whether the requirement is an internal one or external;
- req_strength: soft or hard requirement;
- req_category: which requirement class that the requirement belongs to, e.g. functional requirement, cost requirement, environmental requirement, etc.;

- req_origin: explicit or implicit requirement;
- req_status: whether the requirement is active, suspended, or inactive;
- req_verify_method: the method and procedure of verifying the requirement;
- req_verification_status: whether the requirement has been verified as satisfied, violated, or unknown;
- req_verification_time: the time the requirement is verified;
- req_rationale: the rationale behind the requirement.

4.3 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 logical sentences. Since requirements have been discussed earlier, in the next (sub)-section we list several examples of constraints that are not (primitive) requirements.

4.3.1 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:

$$weight(p) = \sum weight(p_i). \qquad (5)$$

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: [1]

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

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

1. 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.

several levels of power consumption, depends on the modes of the components are on, e.g. 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.

$$(\forall\ p)\ weight(p) > 0. \qquad (6)$$

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

$$material(Base\_cover) = Cast\_iron \vee material(Base\_cover) = Cast\_steel. \qquad (7)$$

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

$$color(Heavy\_base) = Blue \vee color(Heavy\_base) = White \vee color(Heavy\_base) = Black. \qquad (8)$$

4.3.2 Relationship of Parametric Constraints and Parts

Parametric constraint is a special class of constraint; it is largely concerned in *parametric design*, where an artifact is characterized by a set of parameters and a set of constraints that limit the values of these parameters [9]. As defined, parametric constraints are constraints whose expressions have no variables [9]. According to this definition, the constraints (7) and (8), and that from the primitive requirement (2) are parametric constraints, while (6) and (5) 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 (7) has the domain *{materal(Base_cover)}*, (8) 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 has 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:

$$[para(P) \in domain(R)] \vee (\exists f)\, feature\_of(f, P) \wedge [para(f) \in domain(R)]. \qquad (9)$$

4.3.3 Constraint information

A constraint is associated with the following information:

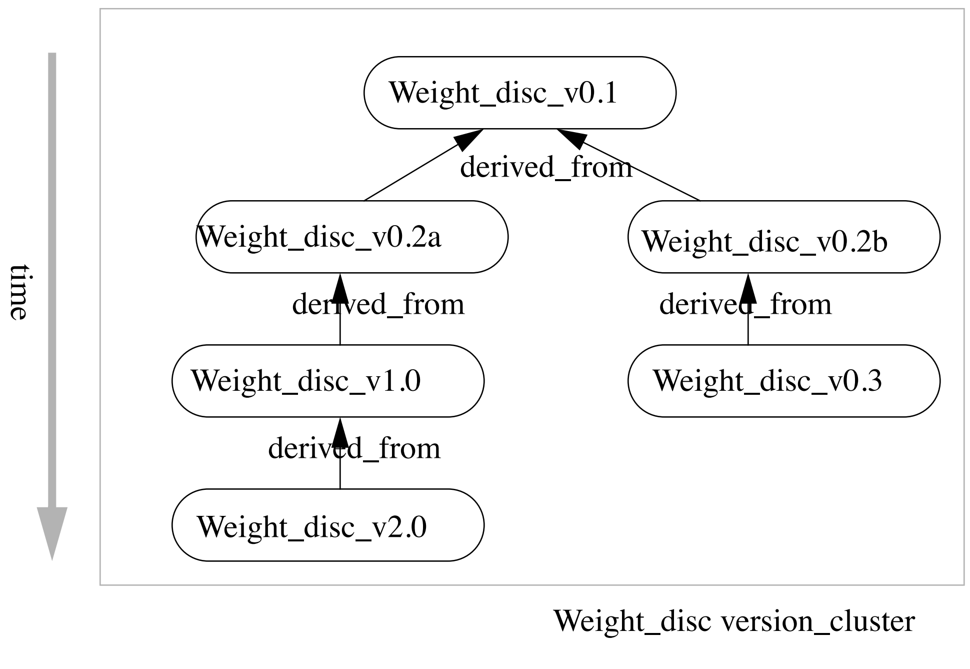
- con_description: descriptive information of the constraint;
- con_doc: the document associated with the constraint containing its textual and graphical description;
- con_exp: the mathematical or first-order description of the constraint;
- con_source: the requirement that the constraint is decomposed from, or physical law etc;
- con_posted_by: the agent who posted the constraint;
- con_posting_time: the date or time when the constraint is posted;
- con_status: whether the constraint is active, suspended, or inactive in the constraint network;
- con_satisfaction_status: whether the constraint is satisfied, violated, or unknown at the con_satisfaction_time;

- con_satisfaction_time: the time that the constraint has the status specified in con_satisfaction_status;
- con_strength: whether the constraint is relaxable (can be given in a relative scale).

5.0 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. Katz [12] defines version as a semantically meaningful snapshot of a design object at some point of time. 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). Figure 7 illustrates the version history of Weight_disc.

FIGURE 7. Versions of Weight_disc



The design starts with Weight_disc_v0.1, from which derives two parallel versions Weight_disc_v0.2a and Weight_disc_v0.2b. They in turn can have dependent versions, Weight_disc_v1.0, Weight_disc_v0.3 and Weight_disc_v2.0 etc. The parallel versions record design alternatives along the design process. This is typically the case in the design of complex and one of the kind artifact—several versions of a design are pursued simultaneously.

The very first version in the version hierarchy is also called *base version*. This is the version of the object that the design begins with. Any object in our ontology (i.e. one of requirements, parts, features and constraints) is either a base version by itself or a version of another object in the ontology. Let *base_version(o)* denote that object *o* is a base version. Then:

$(\forall\ o) \neg\ base\_version(o) \supset (\exists\ o')\ derived\_from(o, o').$

$(\forall\ o)\ base\_version(o) \equiv \neg\ (\exists\ o')\ derived\_from(o, o').$

The second axiom in fact subsumes the first one and gives an definition for *base_version*. For example, Weight_disc_v0.1 is a base version in Figure 7.

A version can only be a derived version of another object. It will be strange if for example Weight_disc_v2.0 is derived from both Weight_disc_v1.0 and Weight_disc_v0.2b. Hence we have the axiom:

$(\forall\ o, o', o'')\ derived\_from(o, o') \wedge derived\_from(o, o'') \supset o' = o''.$

As mentioned, parallel versions record design alternatives and different design paths. The formal definition of parallel version will be illustrated by the following two axioms. The first one says that if two objects are versions of a same object then they are *parallel versions:*.

$(\forall\ o, o')\ [(\exists\ o'')\ derived\_from(o, o'') \wedge derived\_from(o', o'')] \supset parallel\_version(o, o').$

The second one defines the "transitivity" of the *parallel_version* relation:

$(\forall\ o, o', o'')\ derived\_from(o, o') \wedge parallel\_version(o', o'') \supset parallel\_version(o, o'').$

By these definitions, we have Weight_disc_v2.0 is a parallel version of Weight_disc_v0.3 and Weight_disc_v0.2b, etc. Version history and parallel versions are discussed in several papers surveyed in [12], e.g. [13], but in those papers the axioms are not presented.

The concept of version cluster gives an important relationship among versions. A *version cluster* is a set of versions related by the *derived_from* relationship. Each version cluster has a unique name. For example, the versions in Figure 7 all belong to the same version cluster Weight_disc (which is the name of the objects with the version number stripped off). A version cluster is like an abstract class with versions in it as its instances. Hence when we talk about Weight_disc, we mean the version objects in this class. The predicate *version_of(o, cluster)* records that version *o* belongs to the version cluster *cluster*. For example, we have:

version_of(Weight_disc_v0.1, Weight_disc),

version_of(Weight_disc_v0.2a, Weight_disc),

....

Each version has a time at which the version is created. The term *creation_time(o)* denotes the creation time of a version *o*. We can give an order over two different versions according to their creation time. Between two versions o_1 and o_2, we say o_1 is *more recent than* o_2, denoted by $o_2 < o_1$ if and only if $creation\_time(o_2) < creation\_time(o_1)$. (We assume here the order < over two time objects can be understood naturally without explanation.)

There is a constraint that if a version is derived from another, then it must be created later in time. It will be weird if a version is derived from a version created later.

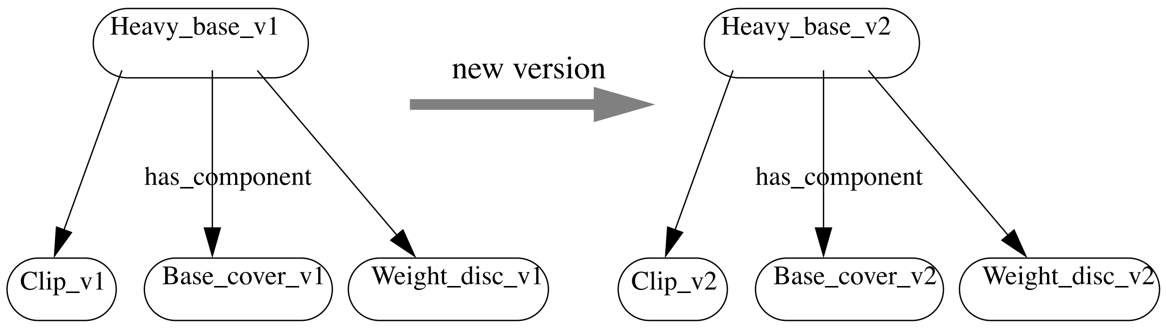
$(\forall\ o_1, o_2)\ derived\_from(o_1, o_2) \supset o_2 < o_1$.

The question such as “What is the most recent version of Weight_disc?” can be answered by finding the object *o* such as the following holds:

$version\_of(o, Weight\_disc) \wedge \neg\ (\exists\ o')\ [version\_of(o, Weight\_disc) \wedge o < o'].$

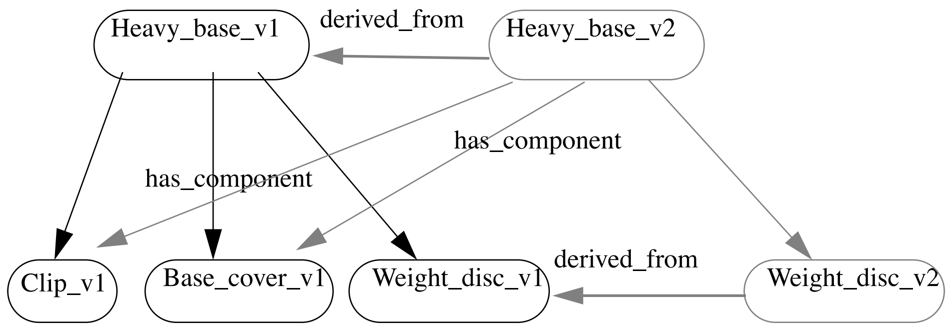
A tricky issue is the version of a *composite* object (a composite object is an object consisting of several sub-objects, e.g. a part that has several sub-components, a requirement that consists of several sub-requirements, a feature that has several sub-features). When one of the sub-objects changed, should a new version of the composite object be created? The answer is yes. In the following we will focus on the maintaining of links between new version of the composite object and the sub-objects. Suppose in the desk spot lamp example, the part Weight_disc has changed and so a new version of Weight_disc is created. As a result new version of Heavy_base is to be created. One possible solution is to make a copy of every sub-components of the Heavy_base so that the new version is “independent of” the old version (Figure 8).

FIGURE 8. Version of Composite Object (1)



This method has to create versions of all sub-objects even if they are not involved in the change, e.g. Clip and Base_over are made copies in the new version even change occurs in Weight_disc only. This incurs substantial duplication if the composite object has many components and the change is limited only on small number of the components. In our KAD system, only those objects that are involved in the changes are versioned (Figure 9). Note that Clip and Base_over are not created new versions and the link of Heavy_base_v2 points to the old versions of them.

FIGURE 9. Version of Composite Object (2)



6.0 References

[1] Black, J. E. AI assistance for requirements management. Concurrent Engineering: Research and Applications (1994) 2, 255-264.

[2] Borgida, A., Greenspan, S. and Mylopoulos, J. “Knowledge Representation as the Basis for Requirements Specification” IEEE Computer, 18:82-91, 1985.

[3] 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.

[4] Dym, Clive L. Engineering Design: A Synthesis of Views. Cambridge University Press, 1994.

[5] 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

[6] 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.

[7] 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

[8] Green, M. (Ed.) Knowledge-aided Design. London UK: Academic Press Ltd. 1992.

[9] 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.

[10]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

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

[12]Katz, R. “Towards a unified framework for version modeling in engineering databases,” ACM Computing Surveys, pp. 375-408, 1990.

[13]Katz, R., Chang, E., and Bhateja, R. “Version modeling concepts for computer-aided design databases”. In Proceedings of the ACM SIGMOD Conference, pp. 379-386, 1986.

[14]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.

[15]Massey, B.S. Measures in Science and Engineering: Their Expression, Relation and Interpretation. Ellis Horwood Limited, England. 1986.

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

[17]Roman, G.-C. A taxonomy of current issues in requirement engineering. IEEE Computer, pp. 14-21, April 1985.

[18]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.