Proceedings of the ASME 2015 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2015
August 2-5, 2015, Boston, Massachusetts, USA

DETC2015-46272

ONTOLOGY-BASED EXECUTABLE DESIGN DECISION TEMPLATE REPRESENTATION AND REUSE

Zhenjun Ming[1], Yan Yan[1], Guoxin Wang[1], Jitesh Panchal[2], Chung Hyun Goh[3], Janet K. Allen[3] and Farrokh Mistree[3]

[1] School of Mechanical Engineering
Beijing Institute of Technology
Beijing, P.R. China, 100081
zhenjun.ming@bit.edu.cn
yanyan331@bit.edu.cn
wangguoxin@bit.edu.cn

[2] School of Mechanical Engineering
Purdue University
West Lafayette, Indiana, USA, 47907
panchal@purdue.edu

[3] The Systems Realization Laboratory @ OU
University of Oklahoma
Norman, Oklahoma, USA 73019
david.goh0819@ou.edu
janet.allen@ou.edu*
farrokh.mistree@ou.edu
*Corresponding author

ABSTRACT

The Decision Support Problem (DSP) construct is anchored in the notion that design is fundamentally a decision making process. Key is the concept of two types of decisions (namely, selection and compromise) and that any complex design can be represented through modelling a network of compromise and selection decisions. In a computational environment the DSPs are modeled as decision templates. In this paper, modular, executable, reusable decision templates are proposed as a means to effect design and to archive design-related knowledge on a computer. In the context of the compromise Decision Support Problem (cDSP) we address two questions:

1. What are the salient features for facilitating the reuse of design decision templates?
2. What are the salient features that facilitate maintaining model consistency when reusing design decision templates?

Here, the first question is answered by the identification of reuse patterns in which specific modifications of the existing cDSP templates are made to adapt to new design requirements, and the second question is answered by developing an ontology-based cDSP template representation method in which a rule-based reasoning mechanism is used for consistency checking. Effectiveness of the ontology-based cDSP representation and reuse is demonstrated for the redesign of a pressure vessel.

KEYWORDS

Design Decision, Decision Support Problem, Decision Template, Reuse Patterns, Ontology, Model Consistency

NOMENCLATURE

DSP	Decision Support Problem
cDSP	compromise Decision Support Problem
sDSP	selection Decision Support Problem
DSIDES	Decision Support in the Design of Engineered Systems (software)
PDSIDES	A Knowledge-Based Platform for Decision Support in the Design of Engineered Systems
parameters	Define the physical attributes of an artifact or a system. The value is fixed during the design solution exploration process.
variables	Define the physical attributes of an artifact or a system. The value is varying during the design solution exploration process.
constraints	Define the geometry, behavior or performance restrictions of a design. Function of variables and parameters. Cannot be violated.
goals	Define the aspirations of designers on a design in terms of geometry, behavior or performance aspect. Function of variables and parameters. Can be under or over achieved.

1. FRAME OF REFERENCE

Increasingly competitive and demanding markets are forcing enterprises to search for the means to decrease the time and cost for product development, while maintaining high quality.

Design reuse, using existing design resources (e.g., standard components, processes, knowledge, etc.) for new product development, is widely used in industry.

The compromise Decision Support Problem (cDSP) is described by Mistree and co-authors in [1]. Reported applications include ships [3-5], aircraft [6-7], vehicles [8], materials [9-10] and thermal systems[11], Stemming from the cDSP, decision templates proposed by Panchal and co-authors [2] provide a means to archive and reuse both the product and associated design process knowledge on a computer. Building on the foundation laid in [1] and [2], we take a step further in developing a Knowledge-Based Platform for Decision Support in the Design of Engineered Systems (PDSIDES). The platform is being designed to facilitate designers to execute, reuse, tailor existing templates and develop new templates.

According to Pahl and Beitz [12], product designs are categorized into three types: a) *Original Design* – New tasks and problems are solved using new or novel combinations of known solution principles, b) *Adaptive Design* – The solution principle remains unchanged, only the embodiment is adapted to new requirements and constraints, c) *Variant Design* – The sizes and arrangements of parts and assemblies are varied within the limits set by previously designed product structures. In the context of PDSIDES, three sorts of activities are identified to support these types of designs within the scope of a single cDSP template: 1) original design – creating new cDSP templates, 2) adaptive design – editing existing cDSP templates, and 3) variant design – editing the value of parameters in existing cDSP templates. Undoubtedly in many engineering design cases, multiple decisions are made and a decision template network is needed, and the basic unit of the network – a single cDSP template – is fundamental. In this paper our focus is on individual cDSP templates and the associated development will contribute as a foundation for the development of cDSP template networks.

In addition to the design activities, another key part of the platform is the user. In PDSIDES, three types of users namely Scientists, Engineers and Designers are identified according to how much knowledge they have about the design domain. In the context of a single cDSP template, **Scientists** are domain experts and responsible for the original design of a template. **Engineers** have sufficient knowledge and experience in a specific domain and are responsible for adapting existing templates for applications that the original templates were not designed for. **Designers** have basic knowledge and little experience in a specific domain, they are template users who vary parameters before executing existing cDSP templates. The approach presented in this paper is mainly focused on facilitating **Engineers** in adapting to existing cDSP templates to meet new design requirements.

In order to realize the adaption of existing cDSP templates to new design requirements on a platform, we address two questions in this paper:

1) *what are the salient features facilitating the reuse of the cDSP templates?*
2) *what are the salient features that facilitate maintaining model consistency when reusing the cDSP templates?*

In the first question, we explore the features possessed by cDSP templates to enable reuse through modification. These features are very important for engineers to know since they need to adapt the cDSP templates to different new requirements according to these features. In the second question, we explore the way to maintain model consistency when the cDSP templates are reused with modifications. Model inconsistency is likely to occur when modifications take place in cDSP templates with a lot of complexity (e.g., tens of variables, parameters, constraints and goals etc.), and when the template modifiers are not the original template creators who have all the knowledge about the templates. The model inconsistency, if ignored, will further influence the correctness of the final design solution generated by the template. Therefore, it is important to maintain model consistency when reusing the templates by modifications.

To answer the preceding questions, the remainder of this paper is organized as follows: the foundation of this paper is briefly introduced in Section 2. The identification of the reuse patterns in which specific modifications are made to meet new design requirements is presented in Section 3. An ontology-based cDSP template representation method and a rule-based consistency checking mechanism is presented in Section 4. A pressure vessel adaptive design validation example is presented in Section 5. In the last section we offer some remarks and enumerate future research opportunities.

2. FOUNDATIONS

The two foundations of this paper are introduced in this section. First, the DSP Technique [13] is used as a formal strategy to model the decisions encountered in engineering designs. Second, design decision templates, specifically cDSP templates, are used to model the decision-centric design process in a reusable and computer-interpretable manner.

2.1 Decision-Based Design, the DSP Technique and Decision Support Problems

In Decision-Based Design, the basic premise is that "the principal role of a designer, in the design of an artifact, is to make decisions" [4]. Design is viewed as a process of converting information into knowledge about the product and decisions are the key markers used to determine the progress of design [14]. Design processes can thus be modeled as sets of decisions. The DSP Technique [3,4,13,15,16] is a framework for design, developed based on this mindset. Mistree and co-authors [4] assert that there are two primary types of decisions: the selection Decision Support Problem (sDSP) [17-19] and the cDSP [1]. A solution to the sDSP represents a preference, based on multiple attributes, for one among several feasible alternatives. The cDSP is about making a trade-off between multiple design objectives, it involves improvement of a feasible alternative through modification. Coupled decisions (including dual coupling and multilevel coupling) [17] can be derived from these basic DSPs. Multiple goals are typically encountered in complex system designs. In this paper we mainly focus on the cDSP that is associated with multiple goals. The word and mathematical formulation of cDSP are shown in Figure 1. The formulation comprises of four keywords: The "Given" keyword captures the information available to designers for decision making. The "Find" keyword embodies information about the system variables which describe the physical attributes of an artifact, and information about the deviation variables which describe the deviations to the targets of system goals. The "Satisfy" keyword embodies the information about the system constraints, variable bounds, and

system goals, with which the feasible design space and aspiration space are determined. The "Minimize" keyword embodies the objective, namely, minimizing the deviation from that which is desired to that which is possible. A tailored computational environment known as DSIDES (Decision Support in the Design of Engineering Systems) [20] has been created as an implementation of the method.

cDSP Word Formulation	***cDSP Mathematical Formulation***
Given Alternative to be improved Assumptions used to model the domain Some helpful relations The system parameter The constraints and goals for the system	**Given** n number of system variables p+q number of system constraints p equality constraint q inequality constraints m number of system goals $g_i(X)$ system constraint function $G_i(X) = C_i(X) - D_i(X)$ $f_k(X)$ function of deviation variables
Find The values of the system variables Values of deviation variables	**Find** X_i System variables i = 1, …, n d_i^+, d_i^- Deviation Variables i = 1, …, 2m
Satisfy System constraints System goals (normalized) Bounds on System Variables	**Satisfy** *System constraints* $g_i(X) = 0$ i = 1, …, p $g_i(X) \geq 0$ i = p+1, …, p+q *System goals (linear, nonlinear)* $A_i(X) + d_i^- - d_i^+ = G_i$ i = 1, …, m *Bounds* $X_i^{min} \leq X_i \leq X_i^{max}$ i = 1, …, n $d_i^+, d_i^- \geq 0;\ d_i^+ \cdot d_i^- = 0$
Minimize Deviation Function	**Minimize** Deviation Function: Archimedean $Z = \Sigma_{i=1}^{m} w_i(d_i^-, d_i^+)$ Deviation Function: Preemptive $Z = [f_i(d_i^-, d_i^+), \ldots, f_k(d_i^-, d_i^+)]$ i = 1, …, m

Figure 1. Word and mathematical formulation of cDSP [1]

2.2 Design decision templates

Panchal and co-authors instantiated the cDSP formulation as problem-independent templates [2]. Through these decision templates, the authors showed the reusability of decision-based design processes for different design problems. Modeling design processes using decision templates is based on three key ideas [21]: (a) design processes can be hierarchically broken down into sub-processes, (b) separation of declarative and procedural information enhances the reusability of design processes, and (c) designing is a decision-centric activity. The basic unit in a decision-centric design process is the modular, executable decision templates.

The modeling strategy of decision templates is analogous to the architecture of a printed wiring board with a number of electronic components, as shown in the cDSP templates in Figure 2. The wiring corresponds to the information flow in a process, the chips correspond to problem specific information, and the board defines the way in which information is transmitted. A primary benefit of this modeling approach is that the resulting reusability extends to both the chips and the board independently. Decision templates are created by developed Extensible Markup Language (XML) schemas which is computer interpretable and extensible. The reusable decision templates for modeling design process enable designers to: (1) specify design processes from reusable modular building blocks, (2) execute design processes based on computer-interpretable building blocks, (3) archive the design processes for reuse, and (4) analyze design processes.

A decision template is a declarative representation of a decision support problem with a predefined standard format. The components of the decision template, namely, Preferences, Goals, Parameters, Variables, Constraints, Driver, Objective, Response, and Analysis, are illustrated in Figure 2. The components are shown as "chips" and the information flow between them is modeled as the "board". In order to create a relevant ontology for the creation and modification of a cDSP template we adapt the definition of the decision template offered by Panchal and co-authors. The definition used in this paper is:

> *A design template is a design decision information structure that comprises of multiple modules with different preset formats and relationships among these modules.*

The "modules" here refer to the "chips" and the "relationships" refer to the "board" in Figure 2. A decision template can be executed only after it has been instantiated with specific design information. In the rest of this paper, when we say a result or a solution generated from a template we mean a specific design case.

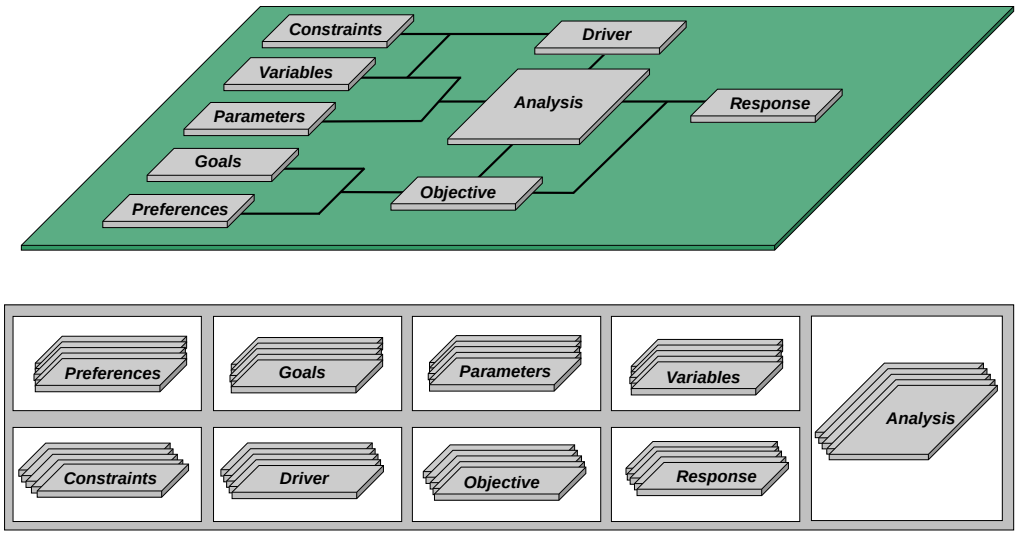

Figure 2. Modular, executable cDSP templates [2]

3. IDENTIFICATION OF THE REUSE PATTERNS FOR THE APLICATION OF DECISION TEMPLATES

What are the salient features that facilitate the reuse of decision templates through modification? To answer this question we turn to the mathematical construct of cDSP. The cDSP is a mathematical construct that embodies the linguistic and mathematical flexibility to address a variety of design problems. In this section, we summarize this flexibility and identify the corresponding reuse patterns in which specific modifications are made to satisfy different requirements.

3.1 Using cDSP to represent existing designs

As adaptive designs are evolved from existing design specifications, we first introduce the representation of existing designs using the cDSP construct. Figure 3 presents a solution space view for a two-dimensional cDSP example, which can be viewed as an existing design. In Figure 3, the coordinate axes (x_1, x_2) stand for the design variables of the problem. C_1, C_2, and C_3 refer to the system constraints that determine the feasible design space together with the four variable bounds. G_1, G_2, G_3, and G_4 represent the four design goals that need to be achieved simultaneously. These goals, with associated monotonicity, form the aspiration space of the design. $d_1^- d_1^+ d_2^- d_2^+ d_3^- d_3^+ d_4^- d_4^+$ stand for the lower and upper deviations of the four goals. Function Z is the deviation function where w_i is the weight of deviation d_i^- or d_i^+. When there is no overlap between the two spaces, designers will need to compromise – finding a solution in the feasible design space that *best* satisfies the goals in the aspiration space (namely minimizing the deviation from that which is desired to that which is possible).

From the solution space view, the cDSP mathematical construct possesses some degree of flexibility to transform and represent different design tasks. This flexibility lies in three aspects:

a) *Solution space dimensions*. A dimension (or a coordinate) of the solution space stands for a series of value options from which designers can choose to characterize a specific aspect of a system or an artifact. The number of Solution space dimensions is determined by how many attributes of a system or an artifact are viewed as variables (whose values can be altered during the solution space exploration process). For example, three variables namely thickness, radius and length are viewed as variables in the design of a pressure vessel in Section 5.1, thus the solution space has a dimension of three. Designers have the freedom to control the solution space dimensions by changing the number of system variables in the cDSP construct. When the number is changed, the previous solution space transforms to a new form and it may lead to different final solutions.

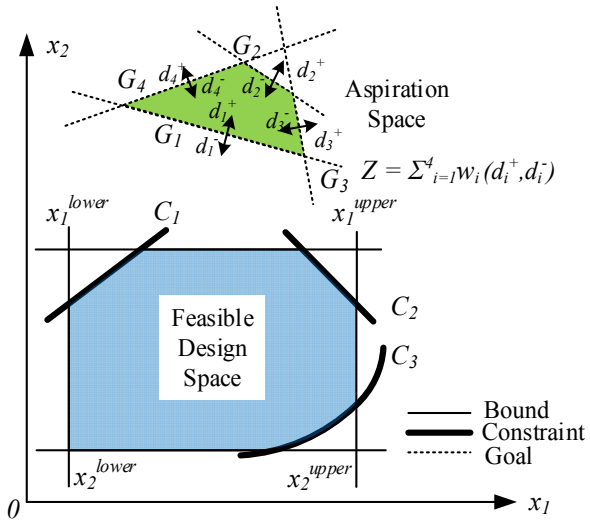

Figure 3. cDSP Solution (2D) space view

b) *Curve shapes*. A curve in the solution space stands for a system constraint or a system goal, and it is one of the boundaries of the feasible design space or the aspiration space. The appearances of feasible design space and the aspiration space are determined by the (boundary) curve shapes. Designers have the freedom to control the curve shapes by the values of system parameters (e.g., control the curve shape of $C(x_1, x_2) = Ax_1 + x_2$ by changing the value of parameter A) in the cDSP construct. When the parameter values are changed,

the previous feasible design space or aspiration space transforms to a new form and it may lead to different final solutions.

c) *The number of curves*. The sizes of the feasible design space and the aspiration space are determined by the number of boundary curves they have. The more boundary curves they have, the smaller size of space is formed, and vice versa. Designers have the freedom to control the number of curves by changing the number of system constraints and system goals in the cDSP construct. When the number of system constraints and system goals is changed, the previous feasible design space and aspiration space transforms to a new form and it may lead to different final solutions.

The flexibility is foundational to identifying cDSP reuse patterns in adaptive designs discussed in Section 3.2.

3.2 cDSP reuse patterns in adaptive designs

A design pattern denotes a formal way of documenting a solution to a design problem in a particular field of expertise [22]. The idea is introduced by architect Christopher Alexander in the field of architecture [23] and has been adapted for various other disciplines, including computer science. Documenting particular problems and associated solutions in patterns can beneficially impose designers' decision about the forms of problems that enable the retrieval and application of solution principles they know. In this paper, analogously, we document specific adaptive design requirements and the corresponding modifications of cDSP as patterns, so as to facilitate **Engineers'** decision about modifications of cDSP in particular adaptive design scenarios. We name the patterns as cDSP reuse patterns, for the reason that the cDSPs associated with designs are reused in these patterns.

A cDSP reuse pattern is a guide for **Engineers** to adapt existing cDSPs to new design scenarios, it consists of two parts: a) a description of design specifications, b) a type of corresponding modifications of cDSP construct. According to the three aspects in which the flexibility of the cDSP mathematical construct lies, as described in Section 3.1, we identify three main cDSP reuse patterns. Two of these patterns are further divided into sub-patterns. Practical adaptive design cases can be of or derived from some individual or combination of the cDSP reuse patterns. Based on the existing design described in Section 3.1, all the identified cDSP reuse patterns (in solution space view) are shown in Figure 4. The introduction of each of the patterns is given as follows:

(1) Pattern 1 – Dimension Change

Design Requirements: Options in some previous design dimensions reduce from multiple (a variable) to one (a constant) or options in some previous design dimensions expand from one (a constant) to multiple (a variable), corresponding adjustment need to be made to the previous design.

cDSP Modification: Convert previous system variables to system parameters (constants), to narrow the associated dimensions (①→②→③, as shown in P1 of Figure 4). Convert previous system parameters to system variables, to enlarge the associated dimensions (③→②→①, as shown in P1 of Figure 4).

(2) Pattern 2 – Size Change

Pattern 2.1 – parameter value change

Design Requirements: New design resources (e.g., materials) with different physical properties (e.g., density, yield stress of a material) are applied in the design, corresponding adjustment need to be made to the previous design.

cDSP Modification: Change the values of system parameters (or variable bounds). For example in P2.1 of Figure 4, parameter A of system goal G_1 is changed to A', parameter D of system constraint C_3 is changed to D', and the value the lower bound of system variable x_1 is changed from K to K'.

Pattern 2.2 – preference value change

Design Requirements: The preferences of goals in multiple objectives design change, corresponding adjustment need to be made to the solution searching strategy of the previous design.

cDSP Modification: As shown in the cDSP mathematical formulation in Figure 1, the deviation function can be formulated in two ways – Archimedean and Preemptive, based on the manner in which importance is assigned to satisfying the goals. In the Archimedean formulation, the deviation function is represented in a weighted sum manner, while in the preemptive formulation it is represented by rank ordering of the goals, see [1]. In this pattern, the value of the weights of the deviations for system goals can be changed if the deviation function is formulated as an Archimedean sum. As shown in P2.2 of Figure 4, the values of w_1, w_2, w_3, etc. can be changed according to the preferences. The preemptive level of deviations can also be changed if the deviation function is formulated in Preemptive fashion.

(3) Pattern 3 – Constraint and Goal Change

Pattern 3.1 – adding new goals and constraints

Design Requirements: Some aspects of the system performance that weren't previously considered are required to pursue and achieve specific target values. Some system behaviors that weren't previously considered must be limited to some degree.

cDSP Modification: Add new system goals and form a new aspiration space (e.g., G_5 is added and a new aspiration space is formed, in P3.1 of Figure 4). Add new system constraints and form a new feasible design space (e.g., C_4 is added and a new feasible design space is formed, in P3.1 of Figure 4).

Pattern 3.2 – deleting existing goals or constraints

Design Requirements: Some aspects of the system performance are not required to be considered any longer. Previous limitation on some system behaviors are not required any longer.

cDSP Modification: Delete some existing goals and form a new aspiration space (e.g., G_2 is deleted and a new aspiration space is formed, in P3.2 of Figure 4). Delete some existing constraints and form a new feasible design space (e.g., C_1 is deleted and a new feasible design space is formed, in P3.2 of Figure 4).

Pattern 3.3 – constraint to goal conversion

Design Requirements: Some previous limitation on the system behaviors are released and required to pursue as system performances. For example, previous limitation on the weight in the design of a car is released and required to be minimized in a new design scenario.

cDSP Modification: Convert previous system constraints to system goals. For example, in P3.3 of Figure 4, C_2 is converted to a system goal and both the feasible design space and aspiration space are reformed.

Pattern 3.4 – goal to constraint conversion

Design Requirements: Some previously pursued system performance objectives are required to be limited to some degree. For example, previous system performance objective "maximizing energy efficiency" in the design of a car is limited as "energy efficiency should not be less than 35%".

cDSP Modification: Convert previous system goals to system constraints. For example, in P3.4 of Figure 4, G_3 is converted to a system constraint and both the feasible design space and aspiration space are reformed.

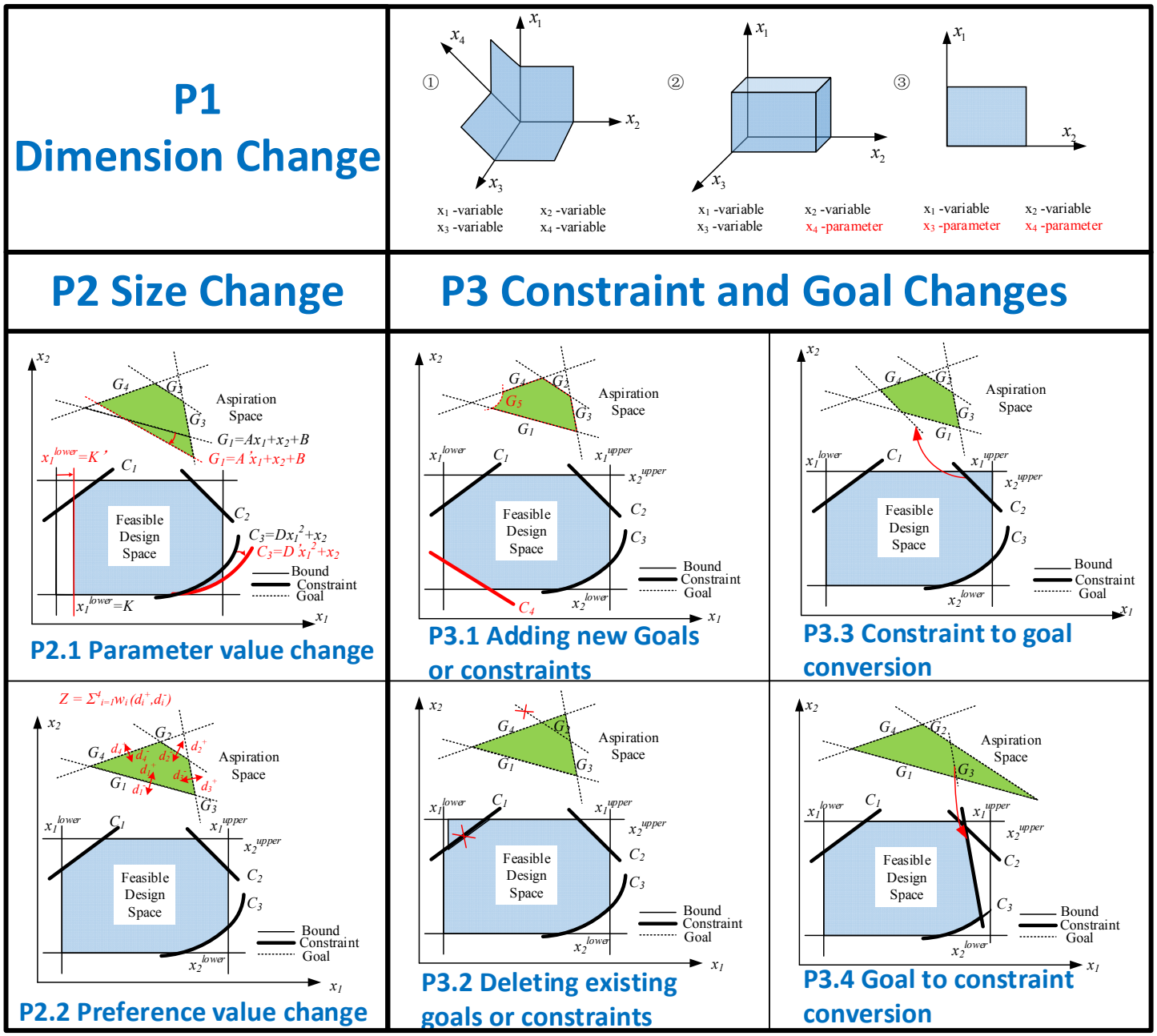

Figure 4 cDSP reuse patterns

3.3 cDSP template operations in the reuse patterns

The modular, executable cDSP templates, with the separation of declarative and procedural information in product design, and with the modularization of declarative information, significantly reduce the complexity and increase the reusability of design a process. In the computational environment, modifications of the cDSP in the reuse patterns can be implemented by specific operations of the cDSP templates. In the operations, information contained in the template is modified while the template itself may remain unchanged. In Table 1, all the possible template operations mapping to the reuse patterns identified in the previous section are presented. The left and the right columns in Table 1 are the reuse pattern list and cDSP template module list respectively, and the template operations mapping to related reuse patterns are illustrated in the middle column. A specific adaptive design can be implemented by any single type or combination of these operations.

The five modules, namely, variable, parameter, goal, constraint and preference listed in Table 1, are the active modules in the reuse patterns, because they are frequently modified and reused in adaptive designs. The rest of the modules in the cDSP template are relatively inactive since they are mainly responsible for exchanging information with the external computational environment and basically remain the same as what they were originally configured.

Table 1. Template operations in reuse patterns

Reuse patterns		Module Operations	cDSP template (active) modules
P1		Conversion between ① and ②	① Variable ② Parameter ③ Constraint ④ Goal ⑤ Preference
P2	P2.1	Value change in ① and (or) ②	
	P2.2	Value change in ⑤	
P3	P3.1	Items adding in ③ and (or) ④	
	P3.2	Items deleting in ③ and (or) ④	
	P3.3	Converting ③ to ④	
	P3.4	Converting ④ to ③	

4. ONTOLOGY BASED CDSP TEMPLATE REPRESENTATION AND CONSISTENCY CHECKING

4.1 Using ontology as the cDSP template representation method

In the cDSP reuse patterns, all the modified information must be verified before computing so as to ensure the correctness of the generated result. While the XML schema used in [2] for representing the cDSP templates does not enable verification of model consistency, on the platform **Engineers** have to manually check the model consistency after they modify some content in the XML file. It is a tedious and error-prone process, especially when the problem is very complex and the file modifier (e.g., an **Engineer**) is not the file creator (e.g., a **Scientist**). Thus, there is a need to represent the cDSP templates in a way that is not only computer-interpretable, but also facilitates maintaining model consistency.

In computer science, ontology is defined as "a specification of a conceptualization" [24]. It provides a common vocabulary for the representation of domain-specific knowledge [25]. With good performance in extensibility and computer-interpretability, ontologies are increasingly being applied in complex applications, e.g., for Knowledge Management, E-Commerce, eLearning, or information integration. In addition, ontologies also possess the inferenceability that enables logical inference on facts through axiomatization [26]. That means the consistency of knowledge can be verified by rules. For the above mentioned three features (extensibility, computer-interpretability and inferenceability), it makes ontology a suitable representation method for the cDSP templates.

The two most widely-used ontology modeling paradigms are Web Ontology Language (OWL) and Frames [27]. A detailed discussion of the similarities and differences is presented by Wang and co-authors [27]. One of the primary differences between Frames and OWL is the view of domain: Frames are based on a closed-world assumption in which everything is prohibited until it is permitted, while OWL is based on an open-world assumption in which everything is permitted until it is prohibited [27]. Since the cDSP template is a computational structure within which most of the modules (e.g., parameter, variable, preference etc.) must strictly comply with certain rules (e.g., in the module "parameter", the upper bound must be equal to the lower bound), it is more like a closed world than an open world. So in this paper we use Frames as the cDSP template modeling paradigm. The ontology is developed in Protégé-Frames 3.5 [28] and the consistency checking mechanism is implemented in JessTab, a plugin of Protégé [29].

4.2 Development of cDSP template ontology

4.2.1 Taxonomy

Panchal and co-authors [2] identify nine modules –goals, constraints, parameters, variables, preferences, driver, analysis, objective, and response – to compose a cDSP template. Within these modules it is important to know that the difference between a *constraint* and a *goal* is that whether they can be violated or not. A *constraint* is a function with a target value that cannot be violated while a *goal* is a function with a target value that can be over or under achieved. In the ontology, a universal class *Function* is introduced to generalize the properties of these two modules and to facilitate conversion. Similar case is with the difference between a *parameter* and a *variable* – a *parameter* is a quantity with only one value option while a *variable* is a quantity with multiple value options, and a universal class *Quantity* is introduced in the ontology for the same purpose. It is also important to know that the module *objective* only captures the monotony (e.g., "Minimize", "Maximize", and "Force") and the expression information which are properties of a *goal*. In order to reduce information redundancy, modules *goal* and *objective* are integrated in one class – *Goal* in the ontology. Since there can be multiple template instances (e.g., instances with different preferences for the goals) that can satisfy the requirements in the design of an artifact or a system, a Class *Design* is introduced in the ontology to capture the general design information. Many template instances (design solutions) are evolved from other instances through modification. A class History is used to capture the relevant historical evolutionary information of an instance. This information is very important for a new designer to understand and reuse a design modified by others.

Following are the definitions for the thirteen classes of the cDSP template ontology:

cDSPTemplate – A class that integrates all the template modules and represents the information structure of a design problem.

Design – A class which captures the information that a system or artifact to be designed to fulfill some functional requirements.

History – A class that captures the historical evolutionary information of a template instance if it is derived from another instance.

Function – A class that represents system behaviors and performances. Superclass of Classes Constraint and Goal.

Quantity – A class representing variable and invariable system properties. Superclass of Classes Variable and Parameter.

Constraint – A class that represents a function with a target value that cannot be violated. Subclass of Class Function.
Goal – A class that represents a function with a target value can be under or over achieved. Subclass of Class Function.
Parameter – A class that represents a quantity with a constant value. Subclass of Class Quantity.
Variable – A class that represents a quantity with multiple value options. Subclass of Class Quantity.
Analysis – An interface to analysis code that evaluates the system response to changes in design variables.
Driver – Captures the file that integrates all the modules and generates analysis code.
Preference – Captures designers' preferences regarding the satisfaction of the system goals.
Response – Captures the result returned from a problem solver (e.g., DSIDES).

4.2.2 Slots of classes

The slots for Frames based ontologies are generally divided into two types – data slots and object slots. Data slots are used to link instances to literals (e.g., link a *name* with a data type of 'String' to a *Variable* instance) while object slots are used to link instances to instances (e.g., link a *Variable* instance to a *cDSPTemplate* instance). The data slots and the object slots for the cDSPTemplate ontology is illustrated in Table 3.

Table 2. Data slots

Slot Name	Definition
name	String. Specifies the name of an instance
description	String. Specifies the descriptive information of a design task
quantityType	Symbol. Specifies the type of a *Quantity* (e.g., "SystemVariable", "SystemParameter", and "DeviationVariable")
symbol	String. Specifies the symbol of a *Quantity*
value	Float. Specifies the value of a *Quantity*
unit	String. Specifies the unit of a *Quantity*
lowerBound	Float. Specifies the lower bound of a *Quantity*
upperbound	Float. Specifies the upper bound of a *Quantity*
expression	String. Specifies the expression of a *Function*
functionType	Symbol. Specifies the type of a *Function* (e.g., "SystemConstraint" and "SystemGoal")
monotony	Symbol. Specifies the monotony of a *Function* (e.g., "Minimize", "Maximize" and "Force")
degree	Symbol. Specifies the form a constraint *Function*: "≤", "≥" or "="
sign	Symbol. Specifies whether a constraint *Function* is linear or nolinear
target	Float. Specifies the target value of a goal *Function*
weight	Float. Specifies the weight of a *Preference* in Archimedean form
level	Symbol. Specifies the level of a *Preference* in preemptive Form. 5 is set as maximum level.
codeFileName	String. Specifies the code file name of a *Driver*
codeFileLocation	String. Specifies code file location of a *Driver*
result	String. Specifies the result information of a *Response*
reusePattern	Symbol. Specifies the reuse patterns of an instance of Class *History*
modification	String. Specifies the modifications of an instance of Class *History*

Table 3. Object slots

Slot Name	Definition
elementOf	Specifies the associated *Functions* that has a certain *Quantity* as an element
functionOf	Specifies the associated *Quantities* in a certain *Function*
associatedGoal	Specifies the goal *Function* that a *Preference* is associated with
input	Specifies the input Quantities of an *Analysis*
output	Specifies the output Quantities of an *Analysis*
hasVariable	Specifies the design variables of a *cDSPTemplate*
hasParameter	Specifies the system parameters of a *cDSPTemplate*
hasConstraint	Specifies the system constraints of a *cDSPTemplate*
hasGoal	Specifies the system goals of a *cDSPTemplate*
hasPreference	Specifies the system preferences of a *cDSPTemplate*
hasDriver	Specifies the driver of a *cDSPTemplate*
hasAnalysis	Specifies the analysis of a *cDSPTemplate*
hasDriver	Specifies the driver of a *cDSPTemplate*
hasResponse	Specifies the response of a *cDSPTemplate*
hasTemplate	Specifies the feasible templates that can support a specific instance of Class *Design*
applyTo	Specify an instance of Class *Design* that a template instance can apply to
hasHistory	Specifies an instance of Class *History* that a template instance has
derivedFrom	Specifies an instance of Class *cDSPTemplate* that a template instance is derived from

The general structure of the Frames based cDSP template ontology can be seen in Figure 5, it presents an insight of how the decision related information is represented. In the ontology, Class *Design* interrelates with Class *cDSPTemplate* by Slots *hasTemplate* and *applyTo*. Class *cDSPTemplate* interrelates with the 'module' Classes *Parameter*, *Variable*, *Constraint*, *Goal*, *Preference*, *Driver*, *Response*, and *Analysis* by corresponding Slots hasParemeter, hasVariable, hasGoal, hasPreference, etc. Interrelations between the 'module' classes are also captured by specific object slots such as *functionOf*, *elementOf*, *associatedGoal*, etc. Data properties of each 'module' class are captured by specific data slots. As the superclass of Classes *Variable* and *Parameter*, Class *Quantity* captures all the common data properties by the data slots, and so does Class *Function* which is the superclass of Classes *Constraint* and *Goal*.

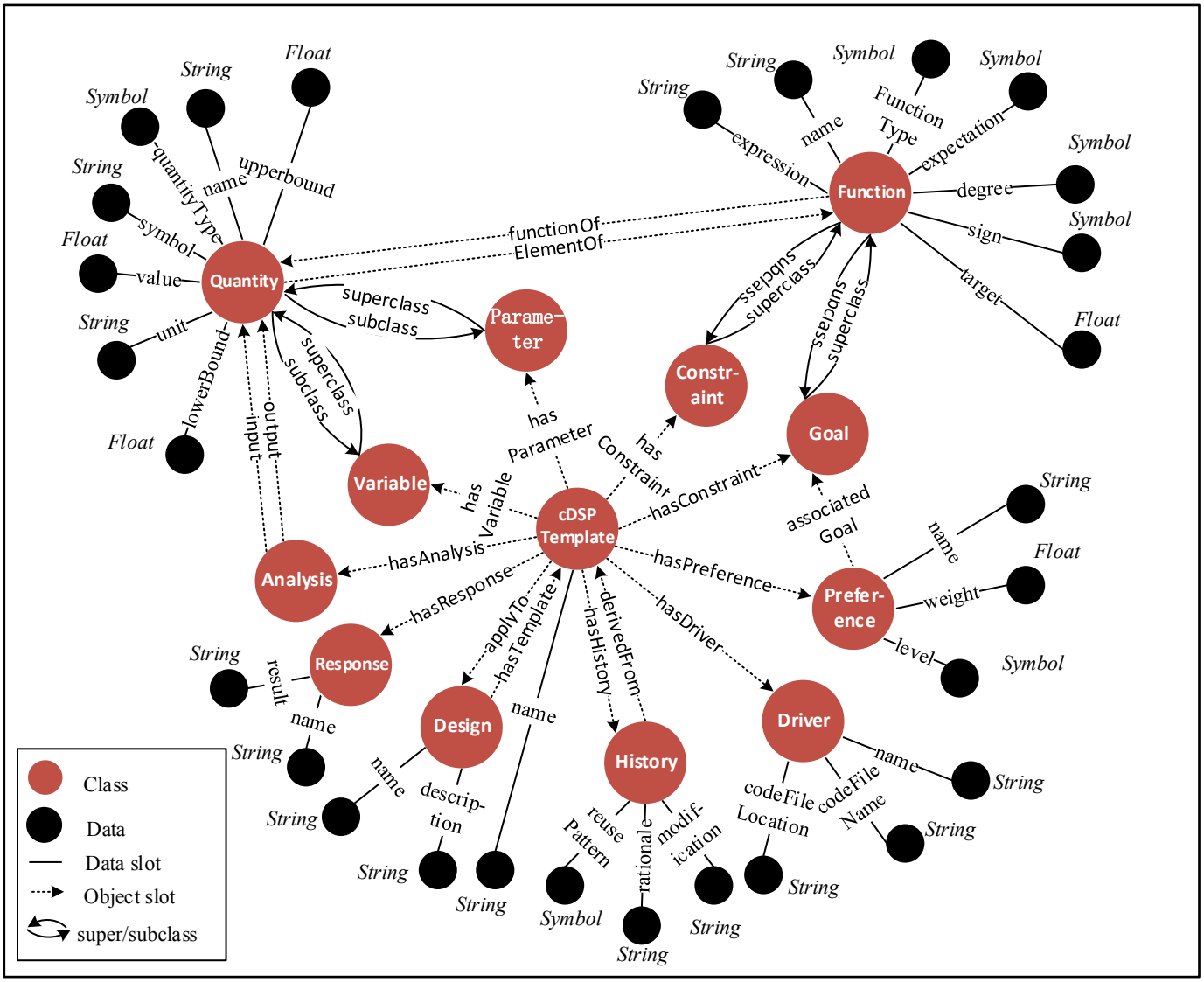

Figure 5. Frames based cDSP template ontology

4.2.3 Instances and the interaction with problem solver

Template instances can be created according to the design problem at hand. Information involved in the items of each keyword of the cDSP formulation is instantiated as instances of the template module classes. The produced module instances are "plugged" into the object slots of Class *cDSPTemplate* to instantiate a cDSP instance. The computation result of this cDSP instance is viewed as a design solution alternative. For example, in the pressure vessel design problem given in Section 5, parameters such as yield strength, stress etc. and variables such as length, radius and thickness, are instantiated as instances of class *Quantity*, and these instances are "plugged" into the object Slots *hasParameter* and *hasVariable* respectively, together with other slots to make a pressure vessel cDSP instance, and result generated by this instance is a solution alternative of the pressure vessel design.

It is necessary for the newly created cDSP instances to be analyzed in a problem solver, and to update themselves with the response from the problem solver. The updated cDSP instances will be documented in the knowledge base as design cases for future reuse. In this research, DSIDES is used as the problem solver. The communication between the knowledge base and DSIDES is implemented by the development of a Protégé Plugin. Through the plugin the template instances in from the

knowledge base are parsed and converted to DSIDES data files. Since it is not our main focus, detailed information about the parsing and converting is not presented in this paper.

4.3 cDSP template model consistency checking

As described in Section 3, much flexibility of modifying the modules (module operations) is allowed in the cDSP templates, while model inconsistency of the modified cDSP templates is a potential risk that cannot be ignored, for it may lead to incorrectness of the final solution. The Webster's dictionary defines inconsistency as "the relation between propositions that cannot be true at the same time, or the lack of harmonious uniformity among parts". In the cDSP template context, model inconsistency is defined as that not all the modules or relation between modules in a template are correct at the same time, due to some modifications. In order to prevent the model inconsistency in the cDSP reuse patterns, we developed a consistency checking mechanism using Rule-Based Reasoning techniques. It is developed and runs in the environment of Jess [30], which is a rule engine for the Java platform and has been integrated to Protégé-Frame as a plugin [29].

Table 4 presents the 12 rules identified for checking the model consistency of the cDSP templates. Rules 1-4 are for checking whether the actual type of some particular instance agrees with the required one. These rules may be triggered in reuse Patterns 1 and 3.4 where conversions between parameters and variables and between constraints and goals take place. Rules 5-8 are for checking whether the numeric value of some particular instance violates the constraint and they may be triggered in Patterns 2.1 and 2.2 where there exist numeric value changes taking place. Rules 9-12 are for checking whether the relation between some particular instances violates the constraint. They may be triggered in pattern 3.1, 3.2, 3.3 and 3.4 where system constraints and goals are modified.

Table 4. cDSP template consistency rules

Rule 1	Check if the type of each instance in the slot "hasVariable" is "SystemVariable"
Rule 2	Check if the type of each instance in the slot "hasParameter" is "SystemParameter"
Rule 3	Check if the type of each instance in the slot "hasConstraint" is "SystemConstraint"
Rule 4	Check if the type of each instance in the slot "hasGoal" is "SystemGoal"
Rule 5	Check if each instance in the slot "hasVariable" complies to: $lowerBound \leq value \leq upperBound$
Rule 6	Check if each instance in the slot "hasParameter" complies to: $lowerBound = value = upperBound$
Rule 7	Check if each instance in the slot "hasPreference" complies to: $0 \leq weight \leq 1$
Rule 8	Check if each instance with the type of "DeviationVariable" complies to: $value \geq 0$
Rule 9	Check if all instances in the slot "hasPreference" comply to: $\Sigma w_i=1$
Rule 10	Check if the two instances with the type of "DeviationVariable" of the same system goal comply to: $d_i^- \cdot d_i^+ = 0$
Rule 11	Check if all the instances of the types "SystemVariable" and "SystemParameter" in the cDSP instance are plugged in the "input" slot of class Analysis
Rule 12	Check if all the instances of the type "DeviationVariable" in the cDSP instance are plugged in the "output" slot of class Analysis

The cDSP template modification and model consistency checking process is illustrated in Figure 6. As shown in the left part of Figure 6. It is a three-step iterative process:

Step 1. Modify cDSP template instances according to design requirements.

Step 2. Check model consistency – if there exists inconsistency then return to the previous step, otherwise go to the next step.

Step 3. Analyze the modified instances in DSIDES. (see Section 4.2.3)

The second step is the crucial step in the process and detailed information about it is presented in the right part of Figure 6. The cDSP instances and the consistency rules are respectively put into the working memory and the rule base, which are the two input interfaces of the Jess rule engine. The pattern matcher in the inference engine applies each instance to the rules one by one, and if any rule is matched, then the action defined in the rule will be put into the agenda and arranged to be executed in the execution engine. For example, when the sum of all the weight of the instances in the Slot *hasPreference* is not 1.0, then Rule 9 will be matched, and action defined in Rule 9 will be put into the agenda and wait for executing. In this research, actions in the rules are defined as warnings of cDSP model inconsistency information.

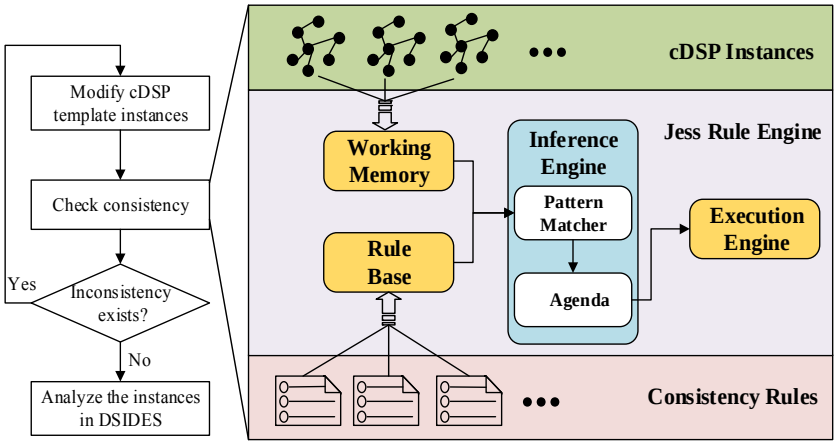

Figure 6. cDSP template modification and model consistency checking process

5. VALIDATION EXAMPLE: ADAPTIVE DESIGN OF PRESSURE VESSEL

A pressure vessel is a closed container designed to hold gases or liquids at a pressure substantially different from the ambient pressure. Since the pressure differential is dangerous, fatal accidents can easily happen. A safe and reliable pressure vessel design is very important. In this section, a cylindrical pressure vessel adaptive design example is presented. The example is about design dimension change, which can be mapped to cDSP reuse pattern 1 described in Section 3.2. The cylindrical pressure vessel adaptive design example is chosen because it is sufficiently complex to demonstrate the effectiveness of ontology-based cDSP template representation and reuse.

5.1 cDSP formulation and ontological template instantiation

Following is a set of previous design requirements of a cylindrical pressure vessel: a) gas contained should be under 1,000 psi; b) the vessel material is UNS G10100 HR with a yield strength of 26, 000 psi; c) the maximum values for the length and height of the vessel are fixed at 60 and 24 in. respectively due to space limitations; d) the radius is required to be at least 5 times greater than the thickness of the vessel; e) industry regulations specify that the design has to comply with a minimum safety factor of 2 for the stresses appearing in both the cylindrical and spherical sections of the vessel; f) maximizing storage volume and minimizing the thickness of the vessel are desired.

Based on the requirements, three design variables namely wall thickness, radius and mid-section length were identified, as shown in Figure 7. And the mathematical cDSP formulation of the problem is presented in Figure 8. Figure 9 presents the cDSP template ontology instance created in protégé based on the formulation. As can be seen in Figure 9, panel (1) is the instance browser where all the instances of a specific class are listed (two instances are listed – Design of a Clindrical Pessure Vssel and Design of a Helical Spring), panel (2) is the instance editor where the detailed information of the selected instance in panel (1) are shown and can be edited by users. Detailed information of the nine slots – *name, hasParameter, hasVariable, has Analysis, hasResponse, hasConstraint, hasGoal, hasPreference, hasDrive*, of the pressure vessel instance is presented in panel. (2) Since most of the slots are object slots except the slot *name*, specific values of these slots are still instances (template module instances). For example the Instances *material yield strength*, *safety factor*, and *vessel internal pressure* in the Slot *hasParameter*. Users can further edit these instances, for example edit the Slot *value* of Instance *material yield strength* and assign a specific number to this slot. All the template operations listed in Table 1 for adapting instantiated templates to new design requirements will take place in panel (2).

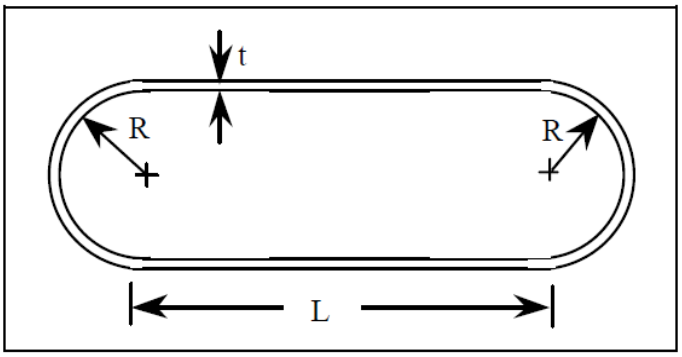

Figure 7. Design variables of a pressure vessel

Given

System parameters

- Material yield strength, $\sigma_y = 26{,}000$ [psi]
- Safety factor, SF = 2 [-]
- Vessel internal pressure, P = 1000 [psi]

Important relations

- Stress in spherical section

  $\sigma_s = P[(R+0.5t)/2t]$ [psi]
- Stress in cylindrical section

  $\sigma_s = P[2R^2 + 2Rt + t^2] / (2Rt + t^2)]$ [psi]

Find

System variables

- Vessel wall thickness, t [in]
- Vessel radius, R [in]
- Vessel mid-section length, l [in]

Deviation variables

- Deviation from target volume, d_1^-, d_1^+ [-]
- Deviation from target thickness, d_2^-, d_2^+ [-]

Satisfy

Variable Bounds

- $0.25 \le t \le 2$, $6 \le R \le 24$, $0 \le l \le 48$ [-]

System constraints

- Maximum stress in spherical section

  $P[(R+0.5t)/2t] \le \sigma_y / SF$ [psi]
- Maximum stress in cylindrical section

  $P[2R^2+2Rt+t^2]/(2Rt+t^2) \le \sigma_y / SF$ [psi]
- Maximum length, $l + 2R + 2t \le 60$ [in]
- Maximum height, $R + t \le 12$ [in]
- Radius and thickness relationship, $R \ge 5t$ [in]

System goals (normalized)

- Maximize volume

  $(4/3\pi R^3 + \pi R^2 l)/30{,}000 + d_1^- - d_1^+ = 1$ [-]
- Minimize thickness

  $0.25/t + d_2^- - d_2^+ = 1$ [-]

Minimize

Deviation function Preemptive formulation

$Z = [(d_1^- + d_2^+)]$

Figure 8. Pressure vessel design cDSP formulation

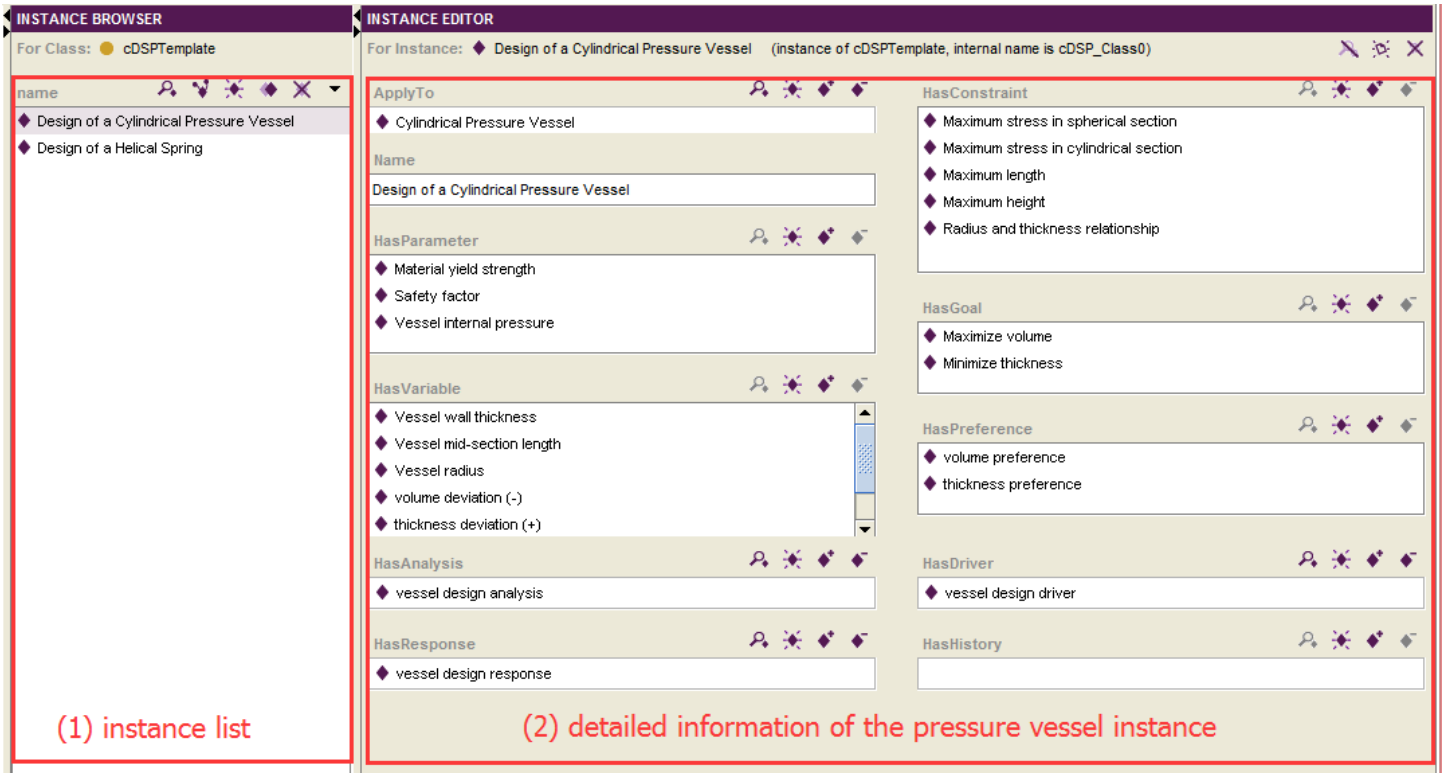

Figure 9. Pressure vessel design cDSP template ontology instance

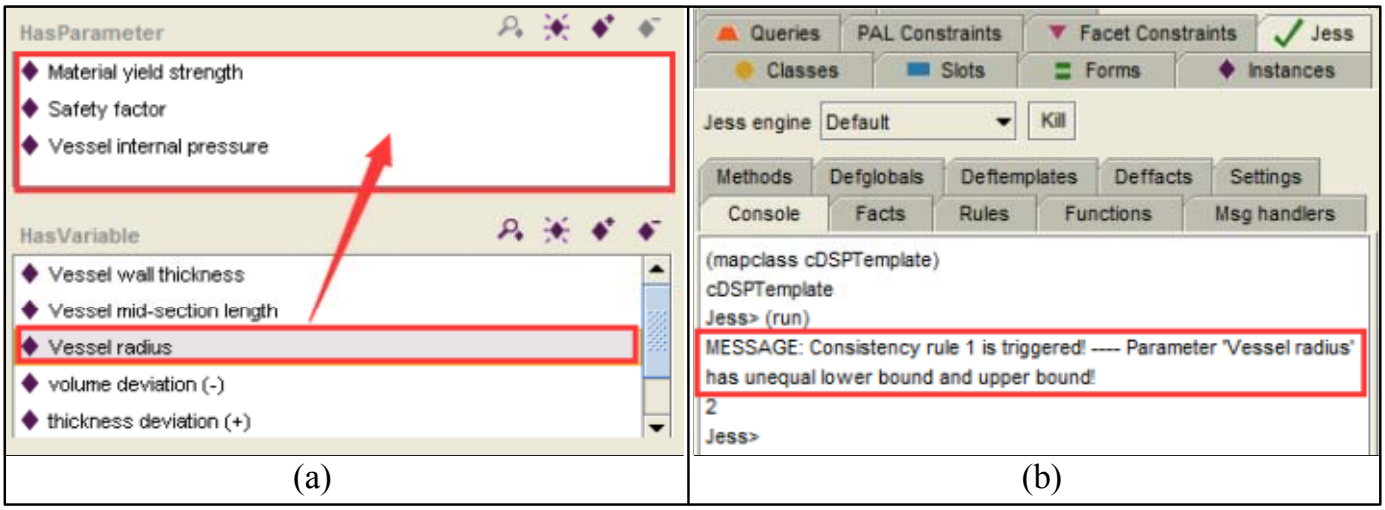

Figure 10. The cDSP template instance a) modification b) model consistency checking

5.2 Adapt the cDSP template instance to new design requirements

A new design requirement of the cylindrical pressure vessel is assumed as follows: due to the manufacturing limitation (the capability of the factory that produces the spherical shells is limited), designers have no choice but an 8-inch spherical radius at hand. And they are required to redesign the pressure vessel considering this limitation.

In this case, the spherical section of the pressure vessel is fixed to constant system parameter because of the aforementioned limitation, and the previous 3-dimension design problem varies to a 2-dimension problem with the Dimension *radius* removed. In order to redesign the pressure vessel, designers usually need to modify the problem analysis code according to this new requirement, and then check the consistency of the codes after modification. Since interdependency exists among different parts in code files, errors are very likely to occur during this process. In the ontology based cDSP template representation context, we used the process presented in Figure 6 to redesign the pressure vessel. Specific steps are as follows:

Step 1. Changed the type of the previous system variable Instance *vessel radius* to "SystemParameter" and modified the previous value to "8.0", then moved Instance *vessel radius* from Slot *hasVariable* to Slot *hasParameter* (see Figure 10.a);

Step 2. Checked consistency. A message from the Jess rule engine came out saying that the values of the lower and upper bound in the parameter "vessel radius" were not equal (see

Figure 10.b). Returned to step 1 and modified the value of both upper bound and lower bound to "8.0". Then checked consistency again, no message came out.

Step 3. Analyzed the modified cDSP template instance in DSIDES and got the result (the value set of all the variables including system variable and deviation variables).

The results for the previous and adaptive pressure vessel designs are shown in Table 5. The result for adaptive design achieved by modifying DSIDES code files which is the traditional reliable way to obtain cDSP results for adaptive design is used as the datum (row ①), to compare and verify the reliability of the corresponding result achieved through modifying the cDSP template instance (row ②). It can be seen that when the previous design variable "radius" is fixed to 8.0 inches (about 2 inches more than the radius of the previous design) according to the new design requirement, the final solution has a larger value of thickness (a 33% increase) and smaller value of length (a 9% decrease), and a reduced deviation of volume target (a 19% decrease) and increased deviation of thickness target (a 27% increase). Also the adaptive design result achieved by modifying the cDSP template instance maintains the same as that achieved by modifying DSIDES code files.

5.3 Qualitative discussion of the result

In the adaptive design example, with one design dimension removed and all other requirements remaining unchanged, the solution need to transfer from balancing among three design variables to balancing between two design variables. The changes in the values of the two variables reflect that transfer. The same result is achieved by modifying DSIDES code files and by modifying the cDSP template instances, suggesting that the ontology-based cDSP template representation method possesses the ability to maintain model consistency when reusing cDSP templates.

In the cDSP template ontology context, the adaptive design of a cylindrical pressure vessel case was implemented only by some simple modifications of the cDSP template slots. No time was spent on understanding the underlying code files that were originally created by others for analysis, and on manually checking whether there exist inconsistencies caused by the modifications in these files. And the final solution can be documented and stored in the knowledge base as an adaptive design case for future reuse. Thus, the primary advantage of the ontology-based cDSP template representation is an efficient (time-saving) and reliable (consistency-ensured) way for design decision related knowledge reuse. The efficiency and reliability improvement brought by this method is particularly beneficial for those **Engineer** users of the platform PDSIDES who are not domain experts and responsible for adapting existing cDSP templates to new design scenarios.

Table 5. Results for the previous and adaptive pressure vessel design cDSP

		System Variable			Deviation Variable			
					Volume		Thickness	
		Thickness (in)	Radius (in)	Length (in)	d_1^-	d_1^+	d_2^+	d_2^-
Previous cDSP		0.481562	6.00511	47.0265	0.792176	0	0.480856	0
Adaptive cDSP *(convert variable "Radius" to parameter)*	①	0.641354	-	42.6987	0.642342	0	0.610200	0
	②	0.641354	-	42.6987	0.642342	0	0.610200	0

① - Modifying of DSIDES code files
② - Modifying the cDSP template instance

6. CLOSURE

In this paper, in order to apply the modular, executable, decision-centric templates to practical engineering design we firstly identify the reuse patterns in which **Engineers** can modify and adapt the decision templates to meet the changing design needs. The reuse patterns are identified according to the flexibility that the cDSP mathematical construct possesses to transform based on different requirements. In the reuse patterns, the modifications of the previous design are mapped to the template operations which are very simple for **Engineers** to handle in adaptive design scenarios. Considering the potential risk of model inconsistency of the decision templates in the reuse patterns where modifications take place, we then develop an ontology-based method in which a rule-based reasoning mechanism is used for consistency checking. Due to cDSP templates' closed-world feature, "Frames" is used as the formalism for developing the ontology. The Jess Rule Engine is used as the reasoning engine for consistency checking. Finally, the effectiveness of the ontology-based cDSP template representation and reuse is tested through a pressure vessel adaptive design example. While the example is simple, the strong handleability and consistency maintainability of the ontology-based executable cDSP templates are demonstrated.

The primary contribution of the approach presented in this paper is an efficient and reliable way to reuse the design decision related knowledge, while the limitation is that it is mainly developed for problems for which single decision making is required and single template is needed. In the future research, we will extend the ontology-based approach to design problems which need multiple decisions and a template network to support the associated process. The research will require: (1) representing the interfaces that connect the templates, (2) maintaining model consistency when topological modifications take place in the networks. The research will be continuously conducted in the context of PDSIDES. We believe that representing design decision templates and the template-based decision network by ontology will enable designers not only to conveniently reuse the design decision related knowledge, but also to ensure consistency when reusing the knowledge.

ACKNOWLEDGEMENTS

Zhenjun Ming and Guoxin Wang gratefully acknowledge the funding (grants 201406030014 and 201406035008) they received from the China Scholarship Council to visit the System Realization Laboratory at the University of Oklahoma. We also gratefully acknowledge NSF grant CMMI-1440457.

REFERENCES

[1] Mistree, F., Hughes, O. F. and Bras, B. A., 1993, "The Compromise Decision Support Problem and the Adaptive Linear Programming Algorithm" in *Structural Optimization: Status and Promise*, pp. 247-286, (M. P. Kamat, Ed.), Washington, D.C.: AIAA.

[2] Panchal, J.H., Fernández M.G., Paredis C.J.J., and Mistree F., 2004, "Reusable Design Processes vis Modular, Executable, Decision-Centric Templates," *AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Albany, NY. Paper Number AIAA-2004-4601.

[3] Mistree, F., Smith, W. F., Bras, B., Allen, J. K. and Muster, D., 1990, "Decision-Based Design: A Contemporary Paradigm for Ship Design," *Transactions, Society of Naval Architects and Marine Engineers*, vol. 98, pp. 565-597.

[4] Mistree, F., Smith, W. F., Kamal, S. Z. and Bras, B. A., 1991, "Designing Decisions: Axioms, Models and Marine Applications," *Fourth International Marine Systems Design Conference*, Kobe, Japan, pp. 1-24.

[5] Smith, W.F. and Mistree, F., 1994, "The Development of Top-Level Ship Specifications: A Decision-Based Approach," *Proceedings 5th International Marine Design Conference,* (Erichsen, S., Gallin, C., Lehmann, E., Leissner, D., Punt, J. eds.), Delft, The Netherlands, May 24-27, pp. 59-76.

[6] Marinopoulos, S., Jackson, D., Shupe, J. A. and Mistree, F., 1987, "Compromise: An Effective Approach for Conceptual Aircraft Design," *AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting*, St. Louis, Missouri, Paper No. AIAA-87-2965.

[7] Lewis, K. and Mistree, F., 1995, "Designing Top-Level Aircraft Specifications: A Decision-Based Approach to a Multiobjective, Highly Constrained Problem," *36th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, New Orleans, Louisiana, April 10-13, 2393-2405. Paper No. AIAA-95-1431-CP.

[8] Erikstad, S. O., Lautenschlager, U., Bras, B., Allen, J. K. and Mistree, F., 1995, "Integrating Robustness into a Multiobjective Space Vehicle Design Process," *AIAA Journal of Guidance, Control and Dynamics*, vol. 18, no. 5, pp. 1163-1168.

[9] Seepersad, C.C., Allen, J.K., McDowell, D.L., and Mistree, F., 2008, "Multifunctional Topology Design of Cellular Material Structures," *Journal of Mechanical Design*, vol. 130, no. 3, pp. 1285-1297.

[10] Seepersad, C. C., R. S. Kumar, J. K. Allen, F. Mistree and D. L. McDowell, 2004, "Multifunctional Design of Prismatic Cellular Materials," *Journal of Computer-Aided Materials Design*, vol. 11, no. 2-3, pp. 163-181.

[11] Smith, W.F., Milisavljevic, Sabeghi, M., Allen, J.K. and Mistree, F., 2015, "The Realization of Engineered Systems with Considerations of Complexity," *ASME Design Automation Conference,* Boston, Massachusetts. Paper Number DETC2015-46211.

[12] Pahl, G., Beitz, W., Feldhusen, J., and Grote, K. H., 2007, *Engineering Design: A Systematic Approach*, Springer.

[13] Muster, D. and Mistree, F., 1988, The Decision Support Problem Technique in Engineering Design. *The International Journal of Applied Engineering Education*, vol. 4, no. 1, pp. 23-33.

[14] Mistree, F., B. A. Bras, W. F. Smith and J. K. Allen, 1996, "Modeling Design Processes: A Conceptual, Decision-Based Perspective," *International Journal of Engineering Design and Automation*, vol. 1, no. 4, pp. 209-221.

[15] Mistree, F., Smith, W. F. and Bras, B. A., 1993, "A Decision-Based Approach to Concurrent Engineering," *Handbook of Concurrent Engineering* (H. R. Paresai and W. Sullivan, Eds.), Chapman & Hall, New York, pp. 127-158.

[16] Bras, B. and Mistree, F., 1991, "Designing Design Processes in Decision-Based Concurrent Engineering," *SAE Transactions Journal of Materials & Manufacturing*, vol. 100, pp. 451-458.

[17] Bascaran, E., Bannerot, R. B. and Mistree, F., 1989, "Hierarchical Selection Decision Support Problems in Conceptual Design," *Engineering Optimization*, vol. 14, pp. 207-238.

[18] Fernández, M. G., Seepersad, C. C., Rosen, D. W., Allen, J. K. and Mistree, F., 2001, "Utility-Based Decision, Support for Selection in Engineering Design, Paper No. DECT2001/DAC-21106," *27th ASME Design Automation Conference (DAC),* Pittsburgh, PA. DETC2001/DAC-21106.

[19] Mistree, F., Lewis, K. and Stonis, L., 1994, "Selection in the Conceptual Design of Aircraft," 5th *AIAA/USAF/NASA/ISSMO Symposium on Recent Advances in Multidisciplinary Analysis and Optimization*, Panama City, Florida USA. pp. 1153-1166.

[20] Reddy, R., Smith, W.F., Mistree, F., Bras, B.A., Chen, W., Malhotra, A., Badhrinath, K., Lautenschlager, U., Pakala, R., Vadde, S., and Patel, P., 1992, *DSIDES User Manual*, Systems Design Laboratory, Department of Mechanical Engineering, University of Houston.

[21] Schnell, A.R., Messer, M.M., Panchal, J.H., Allen, J.K., Hesketh, P.J., and Mistree, F., 2006, "Extending Reusable Decision-Centric Templates to Facilitate Collaboration Using Game Theory," *AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, Portsmouth, VA., Paper No: AIAA-2006-6991.

[22] WIKIPEDIA, 2014, Design Pattern, http://en.wikipedia.org/wiki/Design_pattern, Accessed on January 19, 2015.

[23] Alexander, C., Ishikawa, S., Silverstein, M., and Jacobson, M., 1977, *A Pattern Language: Towns, Buildings, Constructions*, Oxford University Press, ISBN 0-19-501919-9

[24] Gruber T. R., 1993, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, vol 5, no. 2, pp. 199–220.

[25] Noy N. F., and Mcguinness D. L., 2001, "Ontology Development 101: A Guide to Creating Your First Ontology,"

http://protege.stanford.edu/publications/ontology_development/ontology101.pdf, Accessed on January 19, 2015.

[26] Ding, L., Kolari, P., Ding, Z., and Avancha, S., 2007, "Using ontologies in the semantic web: A survey." In. *Ontologies* (*Eds. R. Kishore and R. Ramesh)* Springer US, pp. 79-113.

[27] Wang H. H., Tu S., Noy N., Rector A., Musen M., Redmond T., Rubin D., Tudorache T., Horridge M., and Seidenberg J., 2006, "Frames and OWL Side by Side," http://protege.stanford.edu/conference/2006/submissions/abstracts/7.2_Wang_Hai_Protege_conf.pdf, Accessed April 16, 2015, pp. 1–4.

[28] Stanford Center for Biomedical Informatics Research, 2012, Protege-Frames, http://protegewiki.stanford.edu/wiki/Protege-Frames, Accessed on January 19, 2015.

[29] Henrik Eriksson, 2008, JessTab, http://protegewiki.stanford.edu/wiki/JessTab, Accessed on January 19, 2015.

[30] Ernest Friedman-Hill, 2013, Jess@, the Rule Engine for the JavaTM Platform, http://herzberg.ca.sandia.gov/, Accessed on January 19, 2015.