INTERNATIONAL CONFERENCE ON ENGINEERING DESIGN, ICED'07
28 - 31 AUGUST 2007, CITE DES SCIENCES ET DE L'INDUSTRIE, PARIS, FRANCE

COMBINING CONSTRAINT SATISFACTION AND NON-LINEAR OPTIMIZATION TO ENABLE CONFIGURATION DRIVEN DESIGN

Srikanth Devanathan[1] and Karthik Ramani[1]
[1]Purdue Research and Education Center for Information Sciences in Engineering, Purdue University

ABSTRACT

Product design is inherently a creative process. The designer synthesizes various embodiments of an idea to satisfy constraints and meet customer requirements. In current product design environment, the actual implications of design modifications at the embodiment stage are not clearly understood until the detailed design is completed. This situation presents us with challenges such as representation of the embodiment and its subsequent use in an opportunistic manner for reuse of past knowledge.

In this paper, we model embodiment design as a generalization of the product configuration problem. Product configuration is defined as the process of obtaining valid design instances of a particular product, given its embodiment. Product configuration methods, presented here, aspire to quickly complete important aspects of detailed design of predefined product concepts to provide the designer with performance and design information (configuration solution) to evaluate a particular embodiment. This allows the designer to explore various "what-if" scenarios at the embodiment design stage. Design optimization plays a crucial role in detailed design while constraint-processing techniques dominate product configuration. In this paper, we describe a configuration driven design framework that allows the interplay of optimizers and configurators. The method presented is explained using the case of an automotive cooling circuit.

Keywords: Configuration design, optimization, constraint satisfaction

1 INTRODUCTION AND MOTIVATION

The design process is an iterative map from the customer requirements to the final design of the product [1]. The various elements of a product design are instantiated with minimal conflict during the design process. Function, geometry, behavior, architecture and parameters are some elements of the product design [2]. These elements vary considerably in their brevity. For example, function and behavior are abstract in their description whereas parameters and geometry can be expressed in considerable detail. Further, there exists a definite sequence in which product elements are qualified and instantiated during design to achieve desired objective. For example, geometric parameters can be identified only after a general description of the form of the product. Similarly, performance parameters can be quantified after the behavior is qualified with simulation or testing.

Typical tasks in concept design include functional decomposition, concept selection and synthesis [3]. The product architecture is then designed in the embodiment design stage. During the detailed design stage that follows, elements such as form, manufacturing process and behavior are instantiated. Optimizations are performed in the later phases of the detailed design stage. Optimization of parameter values plays a crucial role in the design process.

Product re-design can be understood as modifications to the elements described above. Moreover, products are re-designed more often than the design of a new product. Therefore, emphasis is placed on techniques that enable re-use and facilitate re-design. Moreover, any new design can be considered a combination of previous designs in a new way. For example, up to 80% of a satellite, a one-of-a-kind-product is designed based on standard components and predefined configurations [4]. Typically, re-designs are achieved through:

1. Modifications to the product architecture: Changes in functional decomposition of the product

can result in modifications to the product architecture. Modifications to the decomposition of the product are caused due to: Change in functional requirement; Combining functions of separate modules into a single module (aggregation); Separating function of a single module into multiple modules (disaggregation)
2. Modifications of form, manufacturing process and performance within the same architecture: This kind of modification may result in introduction of new parameters and changes to the values of certain other parameters.
3. Granular modifications: These are usually small modifications to the values of parameters to improve performance without considerable change to the form or architecture of the product. For example, the changes in the set-up of a racing vehicle according to racing conditions can be considered as granular modifications.

Soininen et al. [5] define the task configuration as "the problem of designing a product using a set of pre-defined components while taking into account a set of restrictions on how the components can be combined". Majority of product configurators are currently used to support sales and quotation tasks using pre-designed components [5]. Product configurator is a powerful tool that can also be used to support the designer during the embodiment design. However, in order to apply product confgurators in a product design scenario, we need to consider the component not as a pre-defined element but as one that represents a *family of variants* or its *product space*. In a re-design that involves changes to the product architecture, a configurator can be used to identify *variant class* or *concept* for each node in the new architecture. The detailed design of the components, i.e. the choice of the individual variant, then completes the design. Furthermore, if the variant class is described in a finite form *a priori*, then the design of the variant can also be automated. In this paper, we report our preliminary work in developing a framework for supporting such a design process, called *configuration driven design (CoDD)*. The overall objective of the framework is shown in Figure 1. The scheme produced in conceptual design is refined using concepts and constraints resulting in a new concept model. The subsequent configuration of the concept model for a given specification produces a valid product instance. This concept model is also stored for later reuse.

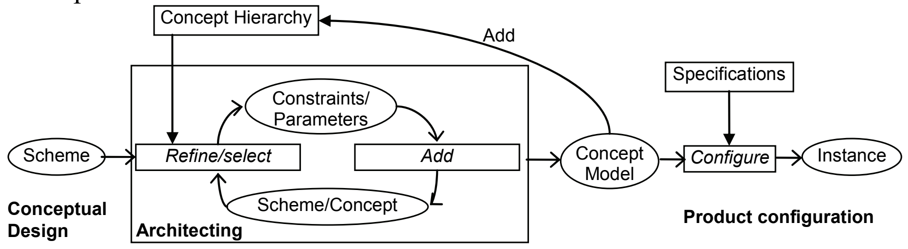


Figure 1: Overall objective of the configuration driven design framework

The following sections deal with the formulation of the embodiment design problem, the model for capturing the design knowledge necessary for configuration and preliminary methods for variant selection.

We propose a configuration driven design framework that
1. Models product configuration involving a generalization of constraint satisfaction and optimization problems.
2. Captures the product space using a constraint based model and allows reuse through configuration.
3. Configures a product using a combination of constraint satisfaction and optimization algorithms.
4. Automatically formulates the design problem based on given specifications, thereby providing flexibility.

Engine air-cooling circuit is introduce as an example in section 3.1.3 to illustrate the use of the concept model in configuration driven design.

2 CONFIGURATION DRIVEN DESIGN

2.1 A model of the design process

Product design processes can be classified into two fundamental types: new design and routine design. In the case of new design, the design commences with a set of functions that are to be satisfied by the product. These overall functions are decomposed into simpler functions or the *working principle*. For example, for an "excavator" the function "lift dirt and load onto a truck" can be decomposed into "pick dirt", "hold dirt", "lift dirt", "move dirt", etc. The designer then identifies a set of *concepts* that perform a combination of the functions. For example, "pick and hold dirt" can be combined and performed by the concept of "bucket". Depending upon the complexity of the interactions between functions, the decomposition results in a *modular* or *integral* structure. Many representations such as paradigm model [6] and systematic model [7, 8] have been suggested in the literature to capture the functional as well as the structural information. Typically, multiple alternatives are generated in which a small set is explored in detail. The initial representation of the structural information along with the engineering specifications is called *scheme* [9]. The structural elements in the scheme can be thought of as the sub-systems in the product. The actual embodiment of the sub-systems, and therefore the system, is not yet fully known. For example, "bucket", "actuator", etc. form the sub-systems for an excavator.

The functional as well as engineering requirements on the sub-systems are used in the design of that sub-system (by repeating a similar procedure for the sub-system, and so on). This process, called *architecting* [10] results in the creation of the *physical architecture* of the product. Hierarchical refinement [11] is a common way of architecting a product. In most design cases, the structural elements (at some level in recurrence) are easily identifiable sub-systems that can be incorporated by routine design using existing knowledge. The embodiment design typically overlaps with the architecting process. During the architecting process, the designer narrows (or qualifies) the type of concept used in the design. In the context of configuration, Sebastian [12] defines *innovative design* as a special case where some of the instances are unknown. Figure 2 shows an example path that the designer of an excavator might take; the thick solid lines indicate the selection of concepts while the thick dashed-lines represent product configuration. A definite classification exists in the concepts that are considered by the designer. Figure 2 also shows this classification hierarchy, for an "actuator". Here, "24V Sol.1" and "24V Sol.2" are instances of "24VDC Solenoid", therefore, lie within the product space. To move down this tree, the designer selects the next node based on perceived attributes such as function, capabilities, behavior, performance, reliability, cost and other requirements. The number of such selections depends on whether the design is a new or a routine design; routine design involves fewer such decisions than new design.

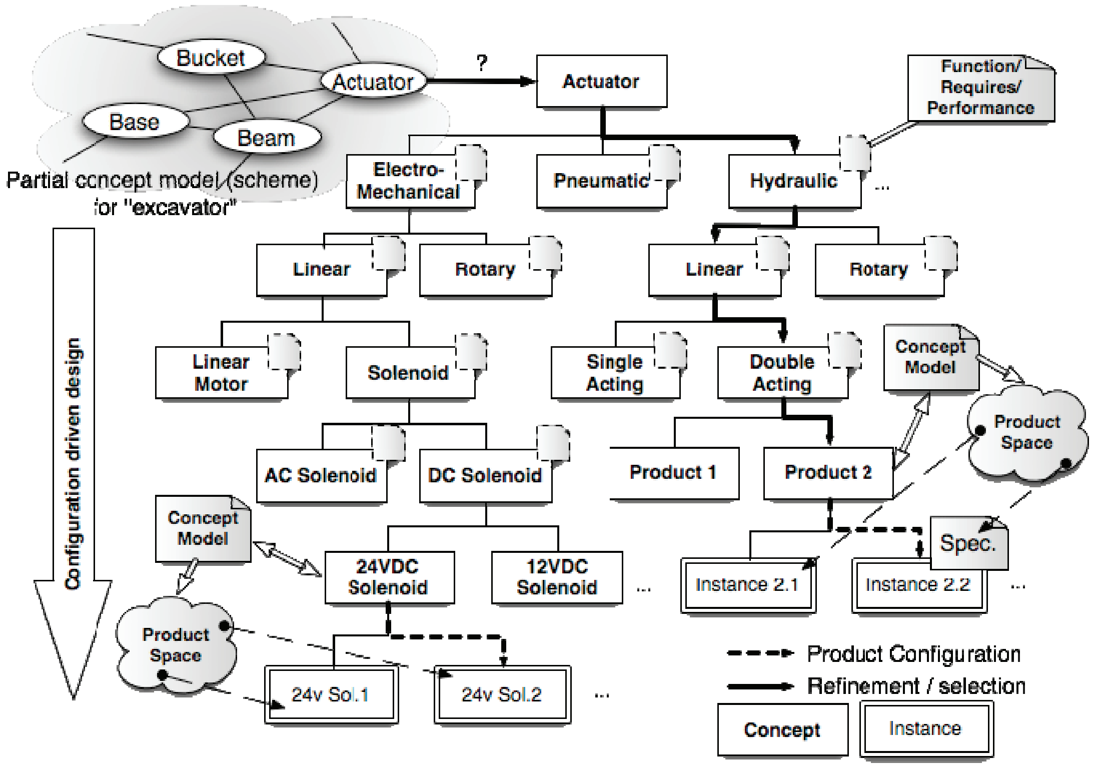


Figure 2: Hierarchy of "Actuator" concepts; The concept model implicitly defines the product space. Product configuration in CoDD is the selection of an instance from the product space.

In configuration driven design framework, each concept in the hierarchy is associated with the notion of the *product space*. The design product space consists of the design space and performance space [13]. The product spaces of the concepts that are lower in hierarchy can be defined with higher precision than those concepts that are higher up. The product space can be represented either explicitly or implicitly. The product space can be explicitly defined by enumerating all possible products. The product space is implicitly defined through constraints, parameters, architecture and domains. In the CoDD framework, the information describing the product space and the knowledge associated with the concept are captured in the *concept model* (see section 3.1.1). The selection of a particular concept modifies the architecture of the product. For example, the selection of a "hydraulic actuator" necessitates the addition of sub-systems such as "hydraulic pump" and "hydraulic control". Further, each selection adds restrictions on other sub-systems in the product. These restrictions are stored in the model of the product in the form of constraints. When the concepts for all the elements in the architecture are selected and the compatibility and other functional constraints are specified, this model is the concept model for the new product concept. The detailed design problem is the instantiation of the various concepts and parameters within this concept without violating any constraint.

The configuration driven design framework is being developed to handle the following issues that arise during such a design process: supporting functional as well as geometric constraints; both continuous and discrete variables/parameters; provision for design optimization and other simulation based design methods.

In general, the steps involved in CoDD are:

1. *A priori* creation of concept model design representation of existing embodiments; this representation is stored in a database to support reuse (See section 5, Future Work). The notion of product space extends across product architecture, i.e., the product space of a system involves the product spaces of some its sub-systems. For standard products such as "bolts" the product space is a collection of discrete points; each point corresponds to a specific instance. The product space is characterized by the concept model associated with it. *The product space is nested within larger product spaces through concept models composed of other concept models within it.* A concept of "solenoid valve" consists of the concepts "solenoid" and "valve"; the product space of the "solenoid-operated valve" is composed of the product spaces of "solenoid" and "valve".
2. Composition of a new design or modification of an existing embodiment through refinement and concept selection. This step is the crucial step in an innovative design. The designer architects the product completely with concept models, assigns constraints and objectives to the model.
3. The selection of a particular product instance based on the objectives and requirements: The designer provides the engineering specifications and uses the concept model to select an instance from the product space. It is typical to involve simulation based design strategies in arriving at an instance.

2.2 Related Work

The focus of the present work is in modeling of a product space using constraints and its subsequent use in configuration. In the context of configuration design of complex products, Feldkamp et al. [14] have proposed a method and software tool, called SyDeR, that combines structural model, a taxonomy based library of solutions, and, constraint propagation techniques. The notion of *ports* is used to encapsulate and model the hierarchical nature of the system being designed. Each port has a predefined direction (either "in" or "out") under SyDeR. The systems being designed are classified as *system types* which are organized in an inheritance called *system taxonomy*. This modeling approach is extended in the CoDD framework to include functional (numerical) constraints within concept models; CoDD also provides flexibility in using *any* interface parameter in *any* direction.

Pahng et al. [15] have proposed a distributed object-based modeling and evaluation (DOME) framework which uses *modules* to represent the product design problem. The design problem is modeled as an aggregation of sub-problems. In the context of DOME, the set of services provided by a module is called an *interface*. Modules that have a common set of services can be interchanged to explore various configuration alternatives. The direction of each interface is also fixed *a priori* in DOME. The emphasis is placed on modeling the product rather than techniques for designing or configuring such products.

In the knowledge aided engineering tool for racing car design, Susca et at. [16] present a model of the system (car) and a coupled solver allowing a designer to quickly evaluate the properties of a car. The independent variables are defined explicitly and the user interface present allows the designer to explore the design by changing these independent variables.
Other systems such as design catalogs [17] have been proposed to store and reuse design models, and also to generate simulation models automatically. Research in design catalogs has mainly consisted of representation and retrieval methods. A Self-configuring component approach [18] has also been proposed to facilitate reusable models in configuration.

3 EMBODIMENT DESIGN PROBLEM

3.1 Design knowledge representation

A general design scenario involves many designers across various economic units such as suppliers and customers. The design tasks are often distributed across the supply chain of the product [19]; the OEM (customer) being in charge of the integration and design of the overall system, while the immediate suppliers providing designs for their respective sub-systems. An important factor in modeling such a relationship is the encapsulation of internal design knowledge while providing flexibility and efficient designs to the "customer". The CoDD framework uses *interfaces* to provide access to the design parameters through which specifications and performance information are communicated. In general, the interface of a concept is the set of design variables that is exposed (by the designer) to other concepts.

3.1.1 Concept model

The embodiment knowledge of a product, called a *concept*, is abstracted into a model to facilitate product configuration. This concept model contains, among others, the product architecture (Φ), compatibility & design constraints (*C*), concept parameters (*P*), objectives (*F*), maps (*M*) and its interface (*I*). In the product architecture, the concepts (primitives and variant classes) that constitute a particular concept are called its sub-concepts. Figure 3 shows the UML representation of the concept. The sub-concepts are stored as a set of identifiers for the primitives and the variant classes that constitute the concept.

In this implementation, we distinguish two kinds of parameters: geometric and functional. Geometric parameters are associated with the form definition of the concept, typically a 3D Computer Aided Design (CAD) file. Functional parameters are parameters that indicate or, in other cases, control the performance of the concept. Three kinds of parameters are available in the current implementation of the CoDD system – scalar, aggregate and reference. Scalar parameters can be of type Integer, Real or String. Aggregate parameters are collection of other parameters. In the current implementation, a concept can also be viewed as an aggregate parameter. Reference parameters "point to" other parameters, and are used to incorporate an interface parameter of a sub-concept with the concept parameters. The interface parameters are a subset of the concept parameters. Compatibility and performance constraints are expressed through the interface parameters of the sub-concepts.

The concept topology indicates the relationships between the sub-concepts. The product topology can be represented as a hyper-graph, called the *concept graph*. The hyper-arcs in the concept graph represent the compatibility constraints between the sub-concepts. The relationships between the parameters are captured through *maps*. Maps can be understood as generalization of design equations. The maps analytically capture the functional knowledge [20] of the product concept. A map is associated with *dependent* and *independent* parameters. The values of the dependent parameters are evaluated by evaluating the map with the values of the independent parameters. Maps, in general, involve algorithmic evaluations such as structural analysis, or tasks involving humans. Further, maps may involve concurrent analyses across multiple domains. In this implementation, we only consider maps that are either evaluated through expressions or by external analysis software. For example, a map involving finite element analysis can be used to evaluate the parameters "*maximum_stress*" and "*deflection*" for a structural concept. The maps in a concept can be visually represented as a directed bipartite graph <Parameters, Maps> as shown in Figure 5. An edge (*p,m*) between the parameter p and map *m* indicates that *p* is an independent parameter in the map *m*, while the edge (*m,p*) indicates that the parameter *p* is a dependent parameter in the map. The concept information is provided to the CoDD system in an XML format.

While a concept model defines the variant family of that concept, a variant is chosen based on a specification. The specification consists of a set of target values for the interface parameters as well as an extra set of constraints. A variant instance is specified by a list of values for *all* the interface parameters as well as the constraints.

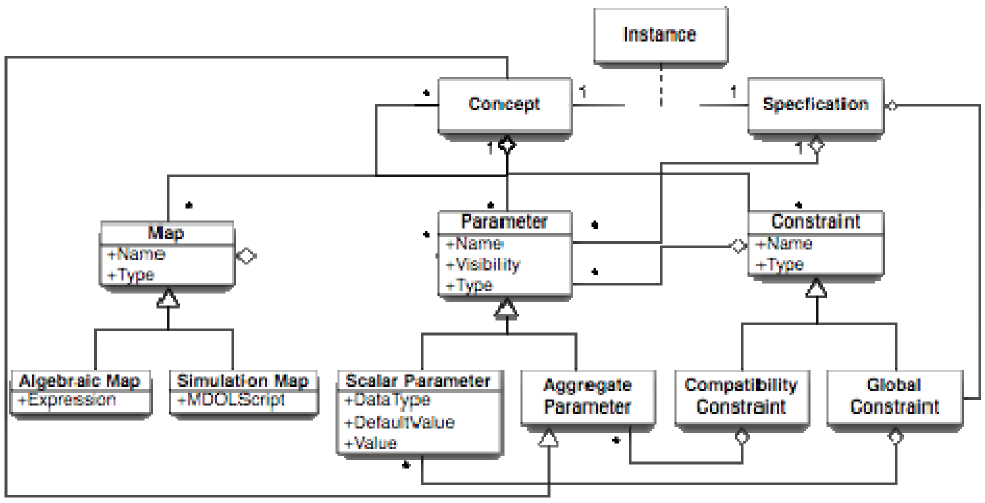


Figure 3: UML Representation of the concept model.

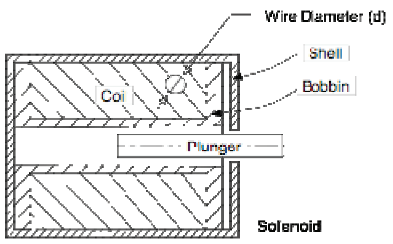
3.1.2 XML Representation of the concept model, Interface and Specification

The concept model information is stored as a loosely typed XML file. Figure 4 shows the XML file for a solenoid-operated valve. The concept name and concept type (“isa”) from the attributes of the concept block. The sub-concepts are listed within the *subconcepts* block. Each sub-concept block is used to provide an alias with the sub-concept identifier and sub-concept type as the attributes. Similarly the parameters, constraints and maps are represented within corresponding blocks. The attributes of a parameter are its visibility (public or private), the descriptive name, the data-type (Real, Integer, String or Aggregate) and the default value. All public parameters are included in the interface of the concept. The optional attribute “references” indicates a reference parameter and the sub-concept parameter as the value. A parameter x of a sub-concept *c* is accessed using the “.” operator as “*c.x*”. The display name is also as an attribute in the constraint and

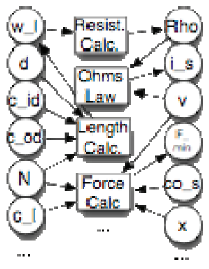
```
<?xml version="1.0"?>
  <concept modelid = “solenoid_valve_1” isa=“hydraulic control” name = “Solenoid Valve”>
  <subconcepts>
    <subconcept modelid = “DC_Solenoid_Model_!” isa = “DC Solenoid”> Solenoid </subconcept>
    <subconcept modelid = “2_way_Valve_1” isa = “2 way valve”> Valve </subconcept>
  </subconcepts>
  <parameters>
    <parameter visibility = “public” dispname = “Voltage (V)” datatype = “real” references =
“Solenoid.OprVoltage” default = “12”> Voltage </parameter>
    <parameter visibility = “public” dispname = “Current (A)” datatype = “real” references =
“Solenoid.MaxCurrent” default = “2”> Current </parameter>
    <parameter visibility = “public” dispname = “Minimum Flow Rate (l/s)” datatype = “real” default = “0”>
MinFlowRate </parameter>
    <parameter dispname = “Solenoid Force” datatype = “real” references = “Solenoid.F_avg” default = “0”>
Force </parameter>
    <parameter visibility = “public” dispname = “Maximum Flow Rate (l/s)” datatype = “real” default = “1”>
MaxFlowRate </parameter>
    <parameter visibility = “public” dispname = “Pressure Drop (Psi)” datatype = “real” default = “12”>
PressDrop </parameter>
    <parameter dispname = “Maximum Flow Rate (l/s)” datatype = “real” default = “1”> MaxFlowRate
</parameter>
...
  </parameters>
  <constraints>
    <constraint dispname = “Force compatibility”> CDATA[! Solenoid.MinForce > Valve.MaxForce !]
</constraint>
    <constraint dispname = “Valve leakage”> CDATA[! Valve.Leakage < 3!] </constraint>
    ...
  </constraints>
  <maps>
    <map dispname = “Length Calculation” type = “algebraic”>CDATA[! Length := Solenoid.Length+Valve.Length
!] </map>
                 <map dispname = “Response Time Calc” type = “simulation”>CDATA[!SimCode MATLAB /
InputFiles trans / FileDescription transcalc.m / FileType: standard / Instructions / require
Solenoid_Force / require ValveSpringStiffness
        SimCodeProcess MATLAB /Program: "./MatlabR11.exe"/ReturnCodes: 1/ End SimCodeProcess ProE / End
SimCode ProE!] </map>
    ...
  </maps>
</concept>
```

Figure 4: XML based concept representation of "Solenoid Valve" used in cooling circuit example

map blocks. The two types of maps - algebraic and simulation are specified within the map block. In this implementation, a simulation (third-party executable) is specified using an MDOL[21] script. In this implementation, the designer provides the MDOL script. MDOL assumes that the executable reads an input file and writes an output file upon evaluation. MDOL records the information necessary



(a)



(b)

(c)

Parameter (dispName)	Data Type	Default	Visibility
w_d (Wire diameter)	real	0.28 mm	private
n (# of turns)	integer	3000	private
rho (Coil resistance)	real	4.29 Ohm	public
c_d(Plunger diameter)	real	7 mm	public
c_l (Plunger length)	real	30 mm	public
F_min (Min. force)	real	0.3 N	public
…			

Figure 5: The Schematic of a solenoid along with maps and parameters of its concept model

for writing an input file with values of the input parameters and reading the output file to obtain the values of the dependant parameters.

3.1.3 Concept example – Automotive charge air cooling circuit

Consider the hydraulic circuit shown in Figure 6. The circuit shown is typically found in automotive engine cooling system. A centrifugal pump driven by the engine circulates the coolant fluid through the heat exchanger called Charge Air Cooler (CAC) where the heat from the supercharged air is transferred to the coolant. The hot coolant is passed through a second heat exchanger (radiator) where the heat from the coolant is transferred to the environment. Varying the rate coolant flowing through the radiator controls the temperature of the supercharged air. The amount of coolant flowing through the radiator is varied by means of an electrically operated 2-way valve as shown. This valve is operated using a pulse-width modulated (PWM) signal. The system consists of four main sub-systems: 2-way valve, the radiator, the engine and the pump. We consider these sub-systems as the sub-concepts for the concept “automotive cooling circuit”. Figure 6 (a) shows the concept graph associated with the concept and some of the compatibility constraints. The solenoid-valve can further be subdivided into a solenoid and the valve.

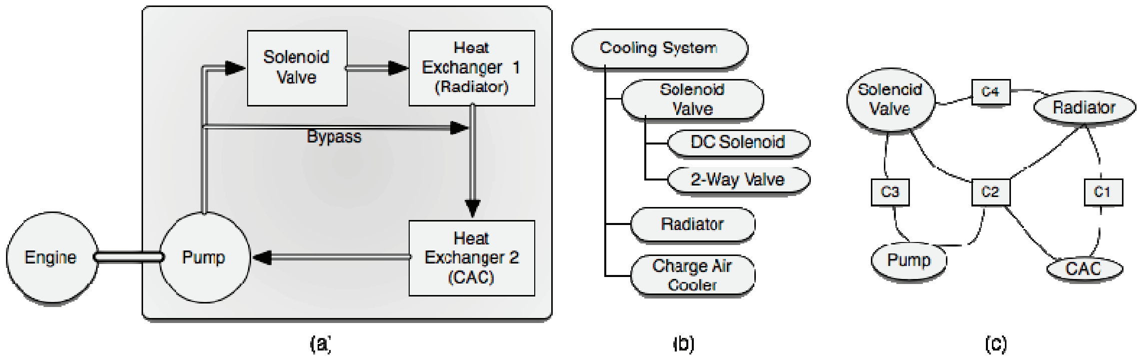


Figure 6: (a) Schematic, (b) architecture and (c) concept graph of the cooling circuit

3.2 Embodiment design problem formulation

The problem formulation is motivated by the fact that a design task, in general, involves both constraint processing and optimization. Therefore it is natural that any formulation of the design problem reduces to a constraint satisfaction problem (CSP) or a classical optimization problem under appropriate restrictions. The embodiment design problem (after configuration) is a collection of sub-problems represented by $\{\langle \Phi, X, D, C, F \rangle_i\}$. Each sub-problem $\langle \Phi, X, D, C, F \rangle$ associated with a concept ϕ is a generalization of optimization and CSP:

Minimize $F = F(\phi) = \{F_i(X)\}, \quad i = \{1,2,...m_1\}$, such that the set of constraints $C = \{G, H\}$ is satisfied, where,

$\Phi = \Phi(\phi) = \{\phi_j\}, j = \{1,2,...m_2\}$, the set of m_2 sub-concepts;

$X = X(\phi) = P \cup \left(\sum_{j=1}^{m_2} I(\phi_j) \right), \quad X \in D$, the design variables;

$D = D(\phi) = D(P) \cup \left(\sum_{j=1}^{m_2} D\big(I(\phi_j)\big) \right)$, the set of domains for the design variables;

$G = G(\phi) = \{g_k(X) \le 0\}, \quad k = \{1,2,...m_3\}$, the set of inequality constraint;

$H = H(\phi) = \{h_l(X) = 0\}, \quad l = \{1,2,...m_4\}$, the set of equality constraints;

$I = I(\phi) = \{I_r : I_r \in X\}, \quad I \subseteq X$, the interface of the concept;

$P = P(\phi) = \{p_s\}, \quad s = \{1,2,...m_5\}, \quad \mathrm{p_s} \in D(p_s)$, the parameters of the concept;

m_1, m_2, m_3, m_4 and m_5 are constants.

A consistent set of solutions to the sub-problems is the solution for the configuration problem. When the problem has exactly one sub problem, for which $(\Phi, G, F) = \emptyset, \quad X = P = \{x_1, x_2, ...x_n\}, \quad D = \{D(x_i)\}, \quad D \not\subset R$ the problem reduces to $\langle X, D, C \rangle$, which is the classical notion of constraint processing. If $D(x_i) \subset R$, the problem reduces to a continuous-CSP (c-CSP). Similarly, when the problem has exactly one sub problem, for which $\Phi = \emptyset, \quad X = P = \{x_1, x_2, ...x_n\}, \quad D = \{(l_i, u_i)\} \subset \mathbf{R}, \quad l_i \le u_i$ the configuration problems reduces to a classical non-linear optimization problem:

Minimize

$F(X)$

where

$X = \{x_1, x_2, ...x_n\}$

subject to

$G = g_j(X) \le 0, \quad j = \{1,2,..m\}$

$H = h_k(X) = 0, \quad k = \{1,2,...,l\}$

side constraints

$D \equiv l_i \le x_i \le u_i \quad \forall i$

Under this formulation, product configuration is a collection of sub-problems. Each sub-problem is the design problem associated with a concept in the product architecture. Each concept model, after architecting, is associated with a solver (configurator) that communicates with other configurators to provide design solutions. A configurator is connected to the configurators associated with the sub-concepts. In this implementation, each configurator configures a single concept for any given specification and external constraint. A specification is a set of target values for a sub-set of interface parameters. Different specifications are obtained by selecting different sub-sets of the interface parameters as well as setting different targets of the selected parameters. *A configurator returns an instance that is closest to the provided specification*.

4 PRODUCT CONFIGURATION IN THE C0DD FRAMEWORK

The strategy for product configuration is a composition of constraint solver and optimizer. In this study, we use a backtracking search for constraint solving and NLPQL[22] for optimization. In the context of this work, product configuration is defined as the *selection of a particular product variant given the concept definition and a specification*. The architecture of a configurator is shown in Figure 7. Figure 8 shows the algorithms for product configuration. The concept XML definition is parsed and the sub-concepts are identified. The configurators for the sub-concepts are created and a connection is established between the configurators. This connectivity among the configurators is topologically identical to the product architecture. Next, the given specification is used to identify the optimization variables and produce an sequence of the maps in a concept. This ordering of the maps is called the solution path [23]. The solution path is the order in which the maps are evaluated, starting with the values of the optimization variables (*OptVars*) and specified interface parameters, resulting in the values of all other parameters in the concept. This evaluation is carried out in step 1 of the *InstantiateSubConcepts* procedure (see Figure 8).

The obtained OptVars are used in the formulation of the optimization problem. Apart from the objectives specified within the concept, a new objective $F_{error}(X)$ is added as:

$$F_{error}(X) = M\left(\sum_{x_i \in Spec}(x_{i,target} - x_{i,evaluated})^2 + \sum_{x_i \in OptVars}(x_{i,guess} - x_{i,evaluated})^2\right),$$

where $x_{i,target}$ is the target specified for the parameter x_i in the concept specification; $x_{i,evaluated}$ is the value of x_i obtained after evaluating the maps starting with the value $x_{i,guess}$ provided by the optimizer; and, M is a large number (exterior penalty method). The procedure *InstantiateSubConcepts* is nested within the optimization loop to obtain a valid combination of sub-concepts. This instantiation is used to evaluate the rest of the concept parameters, the objective functions and the constraint values. Different solution methods can be obtained by modifying the *InstantiateSubConcepts* procedure.

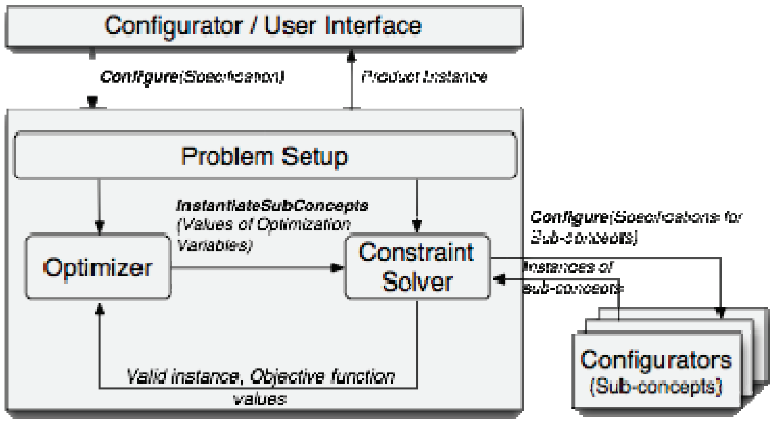

Configurator / User Interface
Configure(Specification)
Product Instance
Problem Setup
InstantiateSubConcepts (Values of Optimization Variables)
Optimizer
Constraint Solver
Configure(Specifications for Sub-concepts)
Instances of sub-concepts
Valid instance, Objective function values
Configurators (Sub-concepts)

Figure 7: Architecture of a configurator in CoDD

The configuration driven design framework is implemented in Java to achieve interoperability and portability. The system allows the designer to create a new concept or modify an existing concept model. The interface also enables to used to conduct many “what-if” studies such as: (a) addition or modification of a constraints, (b) modifications to the specifications, (c) swapping out sub-concepts. By default, the implementation does not execute the optimizer when the designer modifies the concept model; the focus is on obtaining a valid instance. The system first identifies the compatibility constraints that are violated. Only the sub-concepts that are adjacent to these violated constraints are re-instantiated next using the *InstantiateSubConcepts* procedure. Figure 9 shows the main elements of the CoDD system interface: the constraint graph, concept architecture, maps and parameters. A typical result of the *FormulateOptimizationProblem* procedure for the “Solenoid” concept is also shown in Figure 9.

The implemented system provides a mechanism to conduct “what-if” studies by allowing the designer to modify the specifications while the system solves for a valid instance. Moreover, the designer is allowed to substitute one concept model with another for the sub-concepts. Subsequent configuration results in either a valid design along with its performance estimate or a message indicating invalidity. This feedback allows the designer to further refine the concept and complete the design. From our initial experience with the system, we are developing methods to include geometric constraints and CAD solvers into the configuration process. Such methods would allow the designer to initiate configuration from within commercial CAD environments.

An embodiment design support tool that allows a designer to navigate through the concept hierarchy is also currently under development. This tool, when complete, will allow the designer to explore various architectures for the same requirements. The product configuration methods present in this paper would form the final step in the design process.

```
Procedure **Configure**
Require: *Concept, Specification*
Returns: *Concept Instance*
1.  Add constraints from Specification to Concept
2.  Connect to configurators of sub-concepts
3.  Identify optimization variables
4.  Formulate the optimization problem
5.  Formulate the Constraint satisfaction problem
6.  Start Optimizer and Constraint Solver
7.  Return result (Product instance)
```

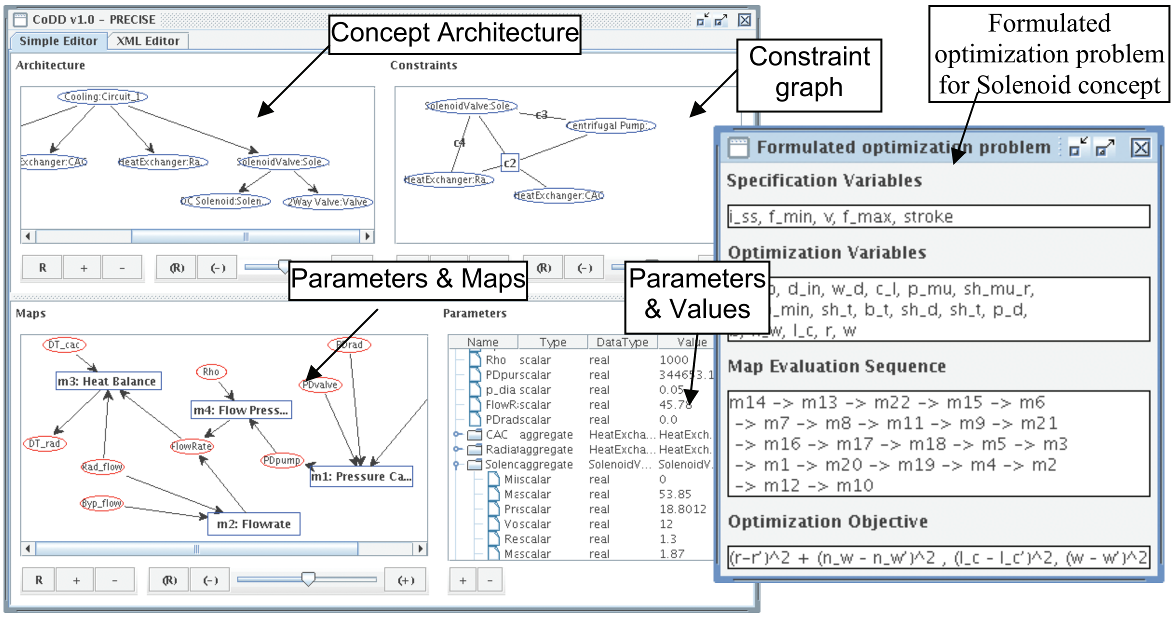
```
Procedure **IdentifyOptimizationVariables**
Require: *Concept, Specification*
Returns: *Optimization variables, Map eval. order*
1.  Initialize MapOrder to empty
2.  OptVars := list of all independent parameters
3.  RestParams := all parameters not in OptVars
4.  For each parameter p in RestParams, do
        If p can be calculated given the values of OptVars
                only, then
            a.  Identify the map m that evaluates p
            b.  Add m to end of end of MapOrder
        3.      else, add p to OptVars
5.  Return OptVars and MapOrder
```

```
Procedure **FormulateConstraintSatisfactionProblem**
Require: *Concept, Specification, OptVars,
SubConceptConfigurators*
1.  CSPConstraints := all constraints that are
    compatibility constraints
    CSPVariables := all sub-concepts
2.  For each x in CSPVariables, do
        SELECT-VALUE (x)  points to
        SubConceptConfigurators[i]::configure
        (SubConceptSpec[i])
```

```
Procedure **InstantiateSubConcepts**
Require: *Concept, SubConceptConfigurators,
         Specification, OptVarValues, OptConstraints*
Returns: *ObjFunctionVals, OptConstraintVals*
  Use OptVarValues and MapOrder to calculate
          values of all concept parameters
          (ConceptParamValues)
  For i = 1 to n, do
  a   Project DesignVarValues, Specifications onto
              interface of sub-concept[i] to get
              SubConceptSpec[i]
  b   AddtlCons := list of constraints that  contain
              interface parameters of sub-concept[i] as
              their only unknowns
  c   Add AddtlCons to SubConceptSpec[i]
  1.  2.      Instance[i] :=  SELECT-VALUE(sub-
      concept[i], SubConceptSpec[i])
  Using the current instantiation, calculate
          ObjFunctionVals and OptConstraintVals
  Return ObjFunctionVals, OptConstraintVals
```

```
Procedure **FormulateOptimizationProblem**
Require: *Concept, Specification, OptVars*
Returns: *ObjFunction, OptConstraints*
1.  OptConstraints := all constraints that are not
    compatibility constraints
2.  For each parameter p in Specification ∪ OptVars,
    do
        If p ∈OptVars and p is a dependent
        parameter in a map m, then add $(p - p_{guess})^2$
        to ObjFunction
        If p ∈OptVars and p∈, then add $(p - p_{target})^2$
        to ObjFunction
3.  Return ObjFunction
```

Figure 8: Listings for major procedures in CoDD implementation



Name	Type	DataType	Value
Rho	scalar	real	1000
PDpur	scalar	real	344653.1
p_dia	scalar	real	0.05
FlowR	scalar	real	45.78
PDrad	scalar	real	0.0
CAC	aggregate	HeatExcha...	HeatExch.
Radiat	aggregate	HeatExcha...	HeatExch.
Solen	aggregate	SolenoidV...	SolenoidV.
Mi	scalar	real	0
Ma	scalar	real	53.85
Pr	scalar	real	18.8012
Vo	scalar	real	12
Re	scalar	real	1.3
Ma	scalar	real	1.87

Formulated optimization problem

Specification Variables

i_ss, f_min, v, f_max, stroke

Optimization Variables

, d_in, w_d, c_l, p_mu, sh_mu_r,
_min, sh_t, b_t, sh_d, sh_t, p_d,
w, l_c, r, w

Map Evaluation Sequence

m14 -> m13 -> m22 -> m15 -> m6
-> m7 -> m8 -> m11 -> m9 -> m21
-> m16 -> m17 -> m18 -> m5 -> m3
-> m1 -> m20 -> m19 -> m4 -> m2
-> m12 -> m10

Optimization Objective

(r-r')^2 + (n_w - n_w')^2 , (l_c - l_c')^2, (w - w')^2

Figure 9: Screenshots of the configuration driven design system

5 CONCLUSION AND FUTURE WORK

In this paper, we have introduced the notion of configuration driven design where the embodiment design is modeled as configuration problem. The general principle of configuration, i.e. selection of appropriate combination of components, is generalized to include the selection of a particular instance from the product space. The product space is characterized implicitly by the constraints defined within the concept knowledge representation. The XML based model to represent the concept knowledge has also been presented. The modular nature of concept model allows nesting of other concepts within a concept and facilitates hierarchical modeling and architecting. The concept model allows encapsulation of concept knowledge using interfaces.

The configuration problem of selecting individual instance form the design space is formulated as a generalization of constraint satisfaction and optimization problems. This formulation reduces to hierarchical optimization technique such as ATC, or CSPs under appropriate restrictions.
The CoDD framework has been implemented in Java. A specification consists of (target) values for the interface parameters and a set of design constraints on the concept. In the CoDD framework, a configurator is attached to a single concept. The configurator configures a concept in collaboration with other configurators attached to other concepts. An example of a cooling circuit for an automobile is presented to illustrate the proposed methods.
Based on the work completed so far, we have identified the following directions for future work:

1. The assumption that a complete and accurate description of the concept knowledge is available before configuration is very unlikely to be encountered in real life. The system should allow the designer to modify the concept knowledge during configuration.
2. The designer deals with many of the constraints and the product architecture through various commercially available CAD packages. The CoDD framework should provide seamless integration with the CAD packages to allow simultaneous solution of the functional constraints and objective during geometric design of components and assemblies.
3. The inclusion of assimilated knowledge, especially rules, is extremely important in a reuse scenario.

ACKNOWLEDGEMENTS
The authors acknowledge the support of Discovery Park Center for Advanced Manufacturing (CAM) for partially supporting the work presented.

REFERENCES

[1] Suh, N.P. *Axiomatic Design: Advances and Applications*. (Oxford University Press, New York, USA, 2001).
[2] Papalambros, P.Y. Analytical Target Cascading in Product Development. *3rd ASMO UK / ISSMO Conference on Engineering Design Optimization*, pp. 3-18Harrogate, New Yorkshire, UK, 2001).
[3] Dieter, G.E. *Engineering Design: A Materials and Processing Approach*. (McGraw Hill, New York, USA, 1999).
[4] Vrinat, M. The case for design and simulation framework. *alpha: The Journal of Virtual Product Development*2005).
[5] Soininen, T., Tiihonen, J., Mannisto, T. and Sulonen, R. Towards a general ontology of configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 1998, **12**, 357-372.
[6] Yoshikawa, H. Design Theory for CAD/CAM integration. *Annals of the CIRP*, 1985, **34**(1), 173-178.
[7] Pahl, G. and Beitz, W. *Engineering Design*. (The Design Council, London, 1984).
[8] Chakrabarti, A. and Bligh, T.P. A scheme for functional reasoning in conceptual design. *Design Studies*, 2001, **22**, 493-517.
[9] O'Sulliven, B.A. Constraint-Aided Conceptual Design. *Department of Computer Science* (National University of Ireland, University College Cork, 1999).
[10] Crawley, E., Weck, O.d., Eppinger, S., Magee, C., Moses, J., Seering, W., Schindall, J., Wallace, D. and Whitney, D. The Influence of Architecture in Engineering Systems. *Engineering Systems Monograph* (Massachusetts Institute of Technology, 2004).
[11] Sriram, R.D. *Intelligent Systems for Engineering*. (Springer-Verlag, London, 1997).
[12] Sebastian, H.-J. Intelligent Systems for Configuration Problems. In Sebastian, H.-J. and Antonsson, E.K., eds. *Fuzzy Sets in Engineering Design and Configuration* (Kluwer Academic Publishers, 1996).
[13] Otto, K. and Wood, K. *Product Design*. (Printice Hall, 2000).

[14] Feldkamp, F., Heinrich, M. and Meyer-Gramann, K.D. SyDeR—System design for reusability. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 1998, **12**, 373-382.
[15] Pahng, F., Senin, N. and Wallace, D. Distribution modeling and evaluation of product design problems. *Computer-Aided Design*, 1998, **30**(6), 411-423.
[16] Susca, L., Mandorli, F., Rizzi, C. and Cugini, U. Racing car design using knowledge aided engineering. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2000, **14**, 235-249.
[17] Kim, J., Will, P., Ling, S.R. and Neches, B. Knowledge-rich catalog services for engineering design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2003, **17**, 349-366.
[18] Germani, M. and Mandorli, F. Self-configuring components approach to product variant development. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2004, **18**, 41-51.
[19] Twigg, D. Managing product development within a design chain. *International Journal of Operations & Production Management.*, 1998, **18**(5), 508-525.
[20] Ishino, Y. and Jin, Y. Acquiring engineering knowledge from design processes. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 2002, **16**, 73-91.
[21] Engineous Inc. www.engineous.com/product_iSIGHT.htm.
[22] Schittkowski, K. NLPQL: A Fortran subroutine for solving constrained nonlinear programming problems. *Annals of Operations Research*, 1985/86, **5**, 485-500.
[23] Salustri, F.A. Towards a Logical Framework for Engineering Design Process. In Cugini, U. and Wozny, M., eds. *From Knowledge Intensive CAD to Knowledge Intensive Engineering*, pp. 213-225 (Kluwer Academic Publishers, 2000).

Contact: Prof. Karthik Ramani
Purdue University
School of Mechanical Engineering
585 Purdue Mall, Room 304
West Lafayette, IN 47906
U.S.A.
Phone: +1 (765) 494-5725
Fax: +1 (765) 494-0539
e-mail: ramani@purdue.edu
URL: http://engineering.purdue.edu/precise