Dealing With Uncertainty in Design and Configuration Problems

Eric Bensana and **Taufiq Mulyanto** and **Gérard Verfaillie** [1]

Abstract. Design is an activity that translate a set of requirements in a feasible product by considering constraints coming from physics, technology avaibility and designer preferences. Configuration is a special case of design in which the product component properties are already predefined. In this paper, the design of a new concepts (or the validation of existing ones) is considered as a configuration problem. In the first part of this paper, we present a generic approach for conceptual design, the earliest phase of design, combining the constraint satisfaction formalism and the object-oriented framework. In the second part, we introduce an approach to deal with uncertainties mostly found during the conceptual design phase. The proposed approach is very easy to implement and guarantees the upper bound of the solution existence probability. The designer can now make his or her decisions not only based on the product performance, but also on the risk associated to that performance.

1 Introduction

Design is an activity that translate a set of requirements in a feasible product which can best satisfy the realization constraints coming from physical phenomenas, the technology availability and designer preferences.

Configuration is a specific task of design. We can see a configuration task as a design of a product given a set of components (possibly sizable) described by a fixed set of properties including informations on the interactions within components. In this paper we restrict the design context in a configuration task.

A *concept* or a product is a solution of a configuration problem as a result of a *product instantiation*. In this paper, we are interested at a system to aid designer in the configuration task using a generic approach.

Our prior internal studies related to the design and sizing of sensing systems [3, 2] showed that the **CSP** framework [12] offers:

- a flexible approach by a natural problem representation using constraints;
- a context free application including an absence of domain type restrictions.

CSP framework is defined by a set of variables, a set of domains each associated to one variable and a set of constraints allowing values combinations within a sub set of variables. Domains can be of all types: symbolic or numeric, discrete or continue. Constraints can be *extensively* defined (list of allowed/prohibited value combinations) or *intensively* defined (relation such as equation). Further, a representation framework such as an object-oriented framework would enhance a design problem representation. We based our proposed generic approach on these two frameworks.

The generic nature of the proposed approach injures consequently its computing power, which limits its use for the *conceptual* design. More advanced phase, such as detailed design, specially in the aircraft design domain, is the realm of parametric optimization [10, 15] even if non classical optimization techniques such as genetic algorithms can be used [1].

Based on the CSP point of view, a number of frameworks have been proposed to deal with configuration problem.

The **Dynamic CSP** framework [14] extends the classical CSP framework to deal with a situation in which the solutions do not need to have the same variables and neither to satisfy the same constraints. The idea is to associate to each variable and each constraint two possible states: active and non-active. States control is done by a set of activation constraints.

The **Composite CSP** framework [19] answers to the same case as previous extension but avoid the use of activation constraints. The idea is to allow a variable to have a sub problem as a value.

The frameworks represented in [17, 22] propose two steps of resolution. The first step consists of determining the final product architecture and the second consists of resolving the corresponding classical CSP problem given the product architecture.

One of the characteristics of design activities, specially during the early stage is the existence of imprecision and uncertainty factors in the problem model and in external input [6, 21]. None of the previous frameworks offers a technique allowing a designer to take into account this uncertainties in his decision making. However, we can find in [13, 18] a probabilistic approach based on **Monte Carlo** method which is time and resource consuming.

In this paper we propose an approach to deal with uncertainty based on CSP framework which guarantees an upper bound of solution existence probability. Our discussions are focused on two points:

1. A description of the generic approach to help designer to structure the design knowledge of a product being considered; this knowledge is expressed in a generic model which is used to generate concepts;
2. A proposition of an approach to deal with uncertainty which guarantees an upper bound of solution existence probability; we will show by a case example how this measure could be an important element of decision.

[1] ONERA/DCSD, 2 av. E. Belin, BP 4025, 31055, Toulouse, France, email: {bensana,mulyanto,verfaillie}@cert.fr

As we are applying the proposed approach in aeronautical domain, examples found in this paper will be on aircraft design problems.

2 Presentation of the generic model

Our generic approach to deal with configuration problem is based on generic model of the product being designed. The generic model combines two frameworks:

- CSP framework with the notions of variables, domains and constraints to formalize the configuration problem;
- Objet-oriented representation with the notion of *objects*, *attributes*, *classes*, *instances* and *inheritance* to structure the configuration knowledge.

2.1 The configuration knowledge

Configuration knowledge, usually related to a given design domain, is structured in classes following an object-oriented representation. Here, a class can be seen as a template of an item. An instance is build upon class description corresponding to an item (final product or a product component).

2.1.1 Classes as generic design knowledge

The knowledge about items is modeled by classes. Typically there is a class for the final product and classes for its components (and recursively). A class can be:

- abstract (instances cannot be built upon directly);
- generic (*e.g. sizeable component*, instance can be built upon, its exact size is not predefined);
- non sizeable (only existing instances of this class are taken into account during the product instantiation).

Simple inheritance is considered, *i.e.* a class has only one mother-class. A class is described by a set of typed attributes.

Attributes

Each attribute is characterized by a definition (mainly its name and type). The domain of values associated with a variable depends on the attribute type. Our system allows the following types:

- `oneof(List)`: `List` is a list of elements of any discrete type (number, atom, string, *etc.*) and the value can be any element of `List`;
- `oneof(Min,Max)`: `Min` and `Max` are two integers, and the value of the variable can be any integer belonging to the `[Min, Max]` interval;
- `instance(Cl1)`: `Cl1` is a class name and the value can be any instance of any subclass `Cl2` of `Cl1` if `Cl1` is abstract or an instance of `Cl1` if not;
- `instance(List)`: `List` is a list of classes name and the value can be any instance of any class in `List`;
- `instances(Min, Max, Cl)` : `Cl` is a class name and the value can be any collection of instances of the class `Cl` whose cardinality belongs to the interval `[Min,Max]`;
- `interval(Min,Max)`: `Min` and `Max` are real numbers and the value can be any subinterval of the interval `[Min,Max]`.

The five first types introduce direct choice points since they all lead to a finite set of possibilities. The last type may also lead to choice points but only when interval splitting is considered. The inheritance property will apply every attribute of a class to its specializations.

For example, the class associated to a wing may has:

- attributes describing wing geometry in real numbers: wing $Area, Span$ and $Aspect\_ratio$, *etc.*;
- attributes determining wing components: flaps, slots, *etc.*

Generic constraints

Generic constraints are expressed at the class level. They define relationships between:

- the values of some attributes of the class; as an example, the three geometric attributes of a wing $Area, Span$ and $Aspect\_ratio$ are related by the constraint: $Aspect\_ratio = Span^2/Area$
- the values of the attributes of some of its components; for example the relation between the wing mass with the mass of its components: $wing\_total\_mass = wing\_mass + flap\_mass + slot\_mass$

In the implementation we have two types of constraints as follows:

- *pre-instantiation* constraints help to prune the search space and avoid to consider wrong options during the product instantiation; there is only a small number of constraints which can be expressed in this way depending on the structural choices for an instance;
- *post-instantiation* constraints are used to verify the instance's attributes consistency during the product instantiation.

Using the inheritance properties, all of constraints in a class are also applied to all of class specializations.

2.1.2 Instances as components

There are two types of instances:

- *generic* instances resulting directly from the instantiation of classes using the product instantiation;
- instances modeling *existing components* or Components On The Shelf (COTS) where all of the attributes have a fixed value (at least those describing its structure); these instances are recorded in a base of components.

After a product instantiation of an item represented by a generic class may result in a generic instance or a component in COTS. But, an item represented as non sizeable class can only be instantiated in one of a component in COTS. COTS can be used to perfectly represent the components non-customizable. In aircraft design, the engines are usually considered as COTS.

2.1.3 Specific constraints as designer's model

A *model* is associated to the class to be instantiated. It determines a set of basic restrictions expressed in unary constraints. The constraints defined in a model takes into account the design requirements and the designer preferences.

A model describes:

- the requirements imposed to the product or its components; in civil aircraft design domain, a requirement can be the typical aircraft flight mission, aircraft capacity, *etc.*
- explicit designer preferences, such as a simple trapezoidal wing plan-form or door type selections.

2.2 The synthesis function

The second main part of our system is the synthesis function. This function is in charge of the product instantiation *i.e.* to instantiate the class corresponding to the product using the available knowledge and product model. Here, we find the connection between CSP formalism and the object-oriented problem representation. During the instantiation of a class, a variable is associated to each attribute. An instance can be considered as a set of variables.

A synthesis function can be expressed as below:

$$Instances = synthesis(Domain, Class, Model, Comps)$$

It builds a set of instances of the class $Class$, with:

- $Domain$: the set of classes in generic knowledge;
- $Model$: a set of constraints which represent the designer preferences and design requirements;
- $Comps$: a set of re-used components in COTS.

The last two entries can be empty. When both are empty, only generic instances will be considered. Other parameters have been defined to limit the search to the first n solutions or within a time limit and to use different levels of constraint propagation.

The set of available classes and COTS implicitly describes the product instantiation search space. Techniques utilized to explore the search space in synthesis function are basically tree search and constraint propagation techniques. During the product instantiation, search in the synthesis function takes into account:

- design choices expressed in class descriptions: choice of a class among several possibles classes defined in a domain attribute, choice of a component in COTS, finite domain attributes, *etc.*
- the choice between reusing an existing component in COTS or building a new generic one;
- other choices related to the management of propagation over intervals.

Depending on the component set and the model, the same synthesis function can be used to:

- *validate* an existing concept using only non-sizeable classes; this can be interesting to verify weather a proposed configuration satisfy the product requirements;
- *define* new concepts : generic classes are authorized and the set of components is reduced to the set of COTS.

2.3 Implementation

We chose a constraint programming language to be our supporting language. The continuous domains is useful to express the sizeable properties on an item. Two languages, although not fully equivalent are selected: `Prolog IV` [4] and `Eclipse` [8]. In our observations, `Prolog IV` is more powerful than `Eclipse` in terms of constraint propagation mechanism, but is much slower as far as pure search is concerned.

The approach combining constraints and objects bears some similarities with the idea implemented in the `Claire` language [11, 7] (notions of parameterized class, abstract class, set type attribute *etc.*). But as `Claire` does not provide, presently, constraints over real intervals, this option has been discarded. In our opinion, it is easier to add basic object-oriented features adapted to our needs in an existing constraint programming language than to develop an efficient constraint propagation mechanism over an existing object oriented language.

To avoid a premature choice between any constraint programming language, we defined a kind of meta-language to express:

- generic constraint expressions;
- generic predicates defining classes and attribute;
- generic predicates accessing attribute values;
- generic predicates to control the product instantiation process.

Both constraint programming language chose do not support a graphical interface. To overcome this disadvantage, web-based utilities such as Netscape, HTML, Javascript, Perl/CGI and Wais are used.

The resulting system, still under improvements, was used for the civil aircrafts design studies [16]. It is currently used for a design of High Altitude Long Endurance Unmanned Aerial Vehicles (HALE UAVs) [5].

UAVs design is an interesting domain for configuration since an UAV can be more or less tailored for the application depending mainly on the type of sensors to be loaded and the flight mission characteristics (range, endurance, altitude, *etc.*).

The current UAV domain is organized such that different aircraft types can be considered: classical configuration plane, flying wing, airplane with a canard wing, *etc.* Different propulsion systems are described : piston engines, turbojets, electric motors with solar arrays or batteries, fuel cells as well as different types of sensors : electro-optical cameras, synthetic aperture radars, spectrometers, data transmission relays. This domain can be easily extended by filling up the domain knowledge by other classes including more general classes.

2.3.1 A small example

Let be the domain considered consists of the following classes:

```
propulsion::
 [[engine,instance(engine)],
  [nb_engines,oneof(1,2)]].

engine::
 [[consumption,interval(0.0,10.0)]].

turbomachine(engine)::
 [[thrust,interval(0.0,100.0)],
  [bypass_ratio,interval(1.0,10.0)]].

turbojet(turbomachine)::
 [[thrust,interval(0.0,10.0)],
  [bypass_ratio,1.0],
  [consumption,interval(0.0,8.0)]].

turbofan(turbomachine)::
 [[thrust,interval(0.0,50.0)],
```

```
  [consumption,interval(0.0,5.0)]].

piston_engine(engine)::
 [[power,interval(0.0,100.0)],
  [r_propeller,interval(0.7,0.9)],
  [d_propeller,interval(0.0,3.0)]].

abstract([engine, turbomachine]).
```

We suppose that the `engine` and `turbomachine` classes are abstract. `turbomachine` and `piston_engine` are classes resulting of a specializations from the class `motor` and `turbojet` and `turbofan` are specialized class from the class `turbomachine`. A specialized class is allowed to have a refined attributes domains from its motherclass as for the class `turbojet`.

Without any specific constraint, six solutions are built by the synthesis function: `s-1, s-2` with turbojet, `s-3, s-4` with turbofan engine and `s-5, s-6` with piston engine.

```
s-1::[[engine,tj-1],[nb_engines,1]]
s-2::[[engine,tj-2],[nb_engines,2]]
s-3::[[engine,tf-3],[nb_engines,1]]
s-4::[[engine,tf-4],[nb_engines,2]]
s-5::[[engine,pe-1],[nb_engines,1]]
s-6::[[engine,pe-2],[nb_engines,2]]
```

with the components:

```
{tj-1,tj-2}::[[consumption,[0.0,0.8]],
              [thrust,[0.0,10.0]]]

{tf-3,tf-4}::[[consumption,[0.0,0.5]],
             [thrust,[0.0,50.0]]]

{pe-1,pe-2}::[[consumption,[0.0,10.0]],
              [power,[0.0,100.0]],
              [r_propeller,[0.7,0.9]],
              [d_propeller,[0.0,3.0]]]
```

If we add a constraint at the `motorisation` stating that if two motors are considered, they can be only turbojets,

```
not((M.nb_engines = 2),
    ((M.engine.class = turbofan)
     ;
     (M.engine.class = piston_engine)))
```

the previous solutions `s-4` and `s-6` will not be built because they will not be consistent.

3 Dealing with uncertainty

After presenting our system in the previous section, in this section we describe our approach to deal with uncertainty in conceptual design, the earliest phase in design activity. We based our proposed approach on CSP framework. We begin by observing two kinds of variable natures.

While representing a design problem as a constraint satisfaction or constraint optimization problem, all the variables do not represent the same thing. Some of them represent possible designer choices. One says that they are controllable. Some others represent uncertainty or imprecision. One says that they are uncontrollable. But such a distinction does not exist in the standard CSP framework. Both are dealt with the same way.

There are at least two extensions of the CSP framework that aim at dealing with uncertainty:

- **Probabilistic Valued-CSP** [20] associates with each constraint (or each combination of values) a probability of existence in the real world (or probability to be forbidden in the real world). The objective is then to find an assignment for all the variables that maximizes its probability to be solution in the real world;
- **Mixed-CSP** [9] explicitly distinguishes controllable and uncontrollable variables. The objective is then to find an assignment for all the controllable variables that is a solution whatever the assignment of the uncontrollable variables is. But, if a probability distribution is available on the domains of all the uncontrollable variables, the objective can be, as in the Probabilistic Valued CSP framework, to find an assignment for all the controllable variables that maximizes its probability to be solution in the real world.

Both extensions only differ in the way of expressing uncertainty: in the first framework, uncertainty is associated with constraints or combinations of values, while variables and domains are the same as in the standard CSP framework; in the second framework, uncertainty is associated with values of uncontrollable variables, while constraints are the same as in the standard CSP framework. At least theoretically, any problem expressed in one of the above frameworks can be expressed in the other.

To deal with uncertainty in design problems, we choose the second framework based on the distinction between controllable and uncontrollable variables. Using the generic model presented previously:

1. we express this distinction in the definition of the classes; then any variable involved in a design problem is either controllable, or not; it suffice to state weather an attribute is controllable or not;
2. we use this distinction to provide the designer with useful information about the consistency probability of the current problem.

We will see that such an extension does not imply any change neither in the generic model, nor in the synthesis function presented above.

3.1 Controllable variables

Here, controllable means that the assignment of a value to the variable is under the control of the designer. We can distinguish three types of controllable variables:

- design variables: they physically describe the designed product; in the example of section 3.4, the attributes representing wing span, the wing area, the lift coefficient, and the aircraft drag due to surface friction are all design variables;
- evaluation variables: they characterize the performance of the designed product; in the same example, the attribute lift-to-drag ratio is an evaluation variable;
- intermediate variables: they are used to simplify the problem expression; still in the same example, the aspect ratio AR is an intermediate variable.

3.2 Uncontrollable variables

On the contrary, uncontrollable means that the assignment of a value to the variable is not under the control of the designer. This may be due to the presence of:

- imprecision or uncertainty in the available design knowledge *i.e.* in item model expressed in classes;
- imperfectly known environment factors; some inputs maybe imprecise or uncertain;
- other designer decisions in a distributed design context.

3.3 Proposed approach

We make the following assumptions:

- uncertainty can be modeled as a probability distribution on the domain of each uncontrollable variable;
- all the uncontrollable variables can be considered as independent.

Using the constraint propagation mechanism, during the product instantiation, controllable variable assignments are propagated in the whole problem and may induce domain reductions on the uncontrollable variables. The smaller the size of the current domains of the uncontrollable variables, the smaller the probability of existence of a real solution in the current problem. We simply propose to use the size of the current domains of the uncontrollable variables to compute an upper-bound on the probability of existence of a real solution in the current problem.

More formally, a $CSP = (V, D, Pr, C, \alpha)$ is defined:

- $V = (Vc \cup Vu)$ where $Vc = \{vc_1, ..., vc_n\}$ is the set of controllable variables, and $Vu = \{vu_1, ..., vu_m\}$ is the set of uncontrollable variables ;
- $D = (Dc \cup Du)$ where $Dc = dc_1 \times ... \times dc_n$, dc_i is the domain associated with vc_i, $Du = du_1 \times ... \times du_m$, and du_j is the domain associated with vu_j;
- $Pr = \{\pi_1, ..., \pi_m\}$ where π_j is, for each uncontrollable variable, the probability distribution associated with du_j;
- C a set of constraints, each of them involving at least one controllable variable;
- $\alpha = \prod_{j=1}^{m} \alpha_j : \alpha_j = f(du_j, \pi_j)$, where $f(du_j, \pi_j) = \int_{du_j} \pi_j(z).dz$ for continuous domains, and $f(du_j, \pi_j) = \sum_{z \in du_j} \pi_j(z)$ for discrete domains.

Under the independence assumption, α is an upper-bound on the probability of existence of a real solution in the current problem. If we make the following assumptions:

- for each uncontrollable variable, the probability to take its value in its initial domain (before synthesis, *i.e.* before constraint propagation and tree search) is equal to 1;
- for each uncontrollable variable, the associated probability distribution is uniform.

α_i is simply the ratio between the size of the current domain of λ_i and the size of its initial domain. α is consequently very easy to compute at any step of the synthesis.

Moreover, by associating each uncontrollable variable with one item (product or component), such an approach can be easily integrated in the object-oriented framework.

3.4 A problem example

We take an example of UAV design for cruising flight. The designer has to determine a wing placement with regard on aircraft body that has good aerodynamic properties by considering the wing geometry uncertainty. In this example the wing placement is represented by the lift coefficient variable.

Here, we consider two sizeable components : aircraft component and wing component, where the wing component is a subpart of aircraft component. Below, we define the `wing` and `aircraft` classes.

The wing class has the following attributes:

- wing span: `span`
- wing surface: `area`
- wing Oswald factor: `oswald`

The Oswald factor is a parameter representing the lift distribution along the wing. It is affected by many other specific wing parameters such as twist angle distribution and wing airfoil distribution. In the early stages of design, this information is not available. One can consider the Oswald factor as an uncontrollable variable taking its value over a given interval. In this example, we take an interval between 0.7 and 0.8. There is no constraint in the `wing` class definition.

The `aircraft` class has the following attributes:

- aircraft aerodynamic property represented by the lift-to-drag ratio: `lift_to_drag`
- aircraft lift coefficient: `c_lift`
- aircraft total drag coefficient: `c_drag`
- aircraft surface friction drag coefficient: `c_drag_fr`
- aircraft wing component: `ac_wing`

The `aircraft` class contains constraints between these variables and the variables in the `wing` class. In this example we assume that the surface friction drag coefficient does not depend on the wing geometry variation.

```
wing::
 [[span, interval(0.0,10.0)],
  [area, interval(0.0,20.0)],
  [unc(oswald), interval(0.7,0.8)],
  [alpha, interval(0.0,1.0)]].

aircraft::
 [[lift_to_drag, interval(0.0,20.0)],
  [c_lift, interval(0.0,1.0)],
  [c_drag, interval(0.0,1.0)],
  [c_drag_fr, interval(0.0,1.0)],
  [ac_wing, instance(wing)],
  [alpha, interval(0.0,1.0)]].
```

The predicate `unc(X)` defines `X` as an uncontrollable variable. The following constraints are expressed at the `aircraft` level:

```
aircraft_constraints(Aircraft)
 {
  Wing = Aircraft.ac_wing
  Sp_2 = power(Wing.span, 2.0)
  AR   = div(Sp_2, Wing.area)
  Cl_2 = power(Wing.c_lift, 2.0)
  E    = Wing.oswald
  K    = product(pi, AR, E)
  CDi  = div(Cl_2, K)
  Wing.c_drag = plus(Wing.c_drag_fr, CDi)
  }
```

`Aircraft` is a variable representing the current instance being built, `power`, `div`, `plus`, `product` belong to the set of basic constraints defined in the language and `pi` refers to the value of π.

Let us assume now that the designer has set some controllable variables and expressed his preferences (in IU metrics) in the following aircraft model:

```
Aircraft.ac_wing.span = 6
Aircraft.ac_wing.area = 4.2
Aircraft.c_drag_fr    = 0.02
Aircraft.c_drag      <= 0.0285
```

Let us suppose that the designer has two design alternatives to compare:

1. `Aircraft.c_lift = 0.4`
2. `Aircraft.c_lift = 0.42`

After running the synthesis function, we find the following results for the lift-to-drag ratio, the Oswald factor and the α parameter:

First alternative:

```
Aircraft.lift_to_drag   in [14.04..14.59]
Aircraft.ac_wing.oswald in [0.7..0.8]
Aircraft.alpha           = 1.0
```

Second alternative:

```
Aircraft.lift_to_drag   in [14.73..14.90]
Aircraft.ac_wing.oswald in [0.77..0.8]
Aircraft.alpha           = 0.3
```

An aircraft has a better aerodynamic property when it has higher lift-to-drag ratio. The designer may then choose the second alternative because of its higher lift-to-drag value, but this solution induces a lower value of the upper-bound on the probability of existence of a solution, *i.e.* a higher risk. Thus, a prudent designer may choose the first alternative instead of the second.

4 Conclusion and perspectives

We have presented a generic approach for configuration and design which associates object and constraint programming technologies. The basic scheme has been extended to deal with some of the uncertainties a designer may encounter in the earliest stages of design.

Possible future developments could consider the introduction of dynamic parameters. Such parameters may also support probabilistic aspects like the probability that a given characteristic takes a given value at a given point of time. Modeling such knowledge will help to address prospective design and give answers to questions such as: is it possible to design a product with a given level of performance within a given time horizon.

Other developments could consider the "compilation" of the knowledge base. Once the domain description is stable, the actual synthesis could be replaced by a more efficient version, by "compiling" the set of classes and producing a constraint program where all design choices are for example expressed as discrete domain variables. This would allow the use of more powerful propagation techniques and helps to define specific design domain configurators.

REFERENCES

[1] T. Barret, G. Coen, J. Hirsh, L. Obrst, J. Spering, and A. Trainer, 'Madesmart : an integrated design environment', in *ASME Design for Manufacturing Symposium*, (Septembre 1997).

[2] C. Barrouil, E. Bensana, C. Castel, L. Chaudron, C. Cossart, R. Manpey, and C. Tessier, 'Programme perception', Rapport final 96-97 RF 2/7996.34 DCSD-T, ONERA/DCSD, (Mars 1998).

[3] G. Bel, M. Barat, and C. Gœuriot, 'Programme PERCEPTION : thème Conception et Dimensionnement', Rapport final 2/7995.02-3575.00, CERT/DERA, (Octobre 1996).

[4] F. Benhamou, 'Lecture notes in computer science', in *Constraint Programming : basics and trends*, ed., A. Poldelski, volume 910 of *Lecture notes in computer science*, chapter Interval constraint logic programming, 1–21, Springer Verlag, (1994).

[5] E. Bensana, 'Etudes conceptuelles d'avions non pilotes : apports de la programmation par contraintes sur les intervalles', in *2eme Congres de la Société Francaise de Recherche Opérationnelle et d'Aide la Décisiosn*, (Janvier 1999).

[6] D.C. Brown, 'Design', in *Encyclopedia of Artificial Intelligence Vol 1*, ed., S.C. Saphiro, 331–339, John Wiley and Sons, New York, (1992).

[7] Y. Caseau and F. Laburthe, 'Introduction to the CLAIRE programming language', Technical report, Département Mathématiques et Informatoique, Ecole Normale Supérieure, France, (1997).

[8] ECRC. Eclipse related papers and reports. `http://www.ecrc.de/eclipse/html/reports.html`.

[9] H. Fargier, J. Lang, and T. Schiex, 'Mixed Constraint Satisfaction : A Framework for Decision Problems under Incomplete Knowledge', in *AAAI-96*, pp. 175–180. AAAI Press, (1996).

[10] I. Kroo, S. Altus, R. Braun, P. Gage, and I. Sobelski, 'Multidisciplanary optimization methods for aircraft preliminary design', *AAIA*, **94**(4325), (1994).

[11] F. Laburthe and Y. Caseau, 'Ecrire du code élégant pour des algorithmes complexes', *Langaes et Modèles à Objets*, (1996).

[12] A.K. Mackworth, 'Consistency in Networks of Relations', *Artificial Intelligence*, **8**(1), 99–118, (1977).

[13] D.N. Mavris, O. Bandte, and D.A. DeLaurentis, 'Robust Design Simulation : A Probabilistic Approach to Multidisciplinary Design', *Journal of Aircraft*, **36**(1), 298–307, (jan 1999).

[14] S. Mittal and B. Falkenhainer, 'Dynamic Constraint Satisfaction Problems', in *8th National Conference of Artificial Intelligence*, pp. 25–32, Boston, MA, (1990). AAAI-90.

[15] F. Morel, 'Multivariate optimization applied to conceptual design of high capacity log range aircraft', *AIAA*, (1994).

[16] T Mulyanto, *Utilisation d'outil de programmation par contraintes pour l'étape conceptuelle de la conception d'avions*, Master's thesis, ENSAE, Aout 1998.

[17] B. O'Sullivan and J. Bowen, 'A Constraint-based Approach to Supporting Conceptual Design', in *Artificial Intelligence in Design '98*, pp. 291–308, Instituto Superior T cnico, Lisbon, (jul 1998).

[18] B. Roth, D. Mavris, and D. Elliott, 'A Probabilistic Approach to UCAV Engine Sizing', in *Joint Propulsion Conference, AIAA98-3264*, Cleveland, OH, (1998). AIAA.

[19] D. Sabin and E.C. Freuder, 'Configuration as Composite Constraint Satisfaction', in *AAAI-96 Fall Symposium on Configuration*, pp. 28–36, (1996).

[20] T. Schiex, H. Fargier, and G. Verfaillie, 'Problème de satisfaction de contraintes valué', *Revue d'Intelligence Artificielle*, **11**(3), 339–373, (1997).

[21] T.W. Simpson, D. Rosen, J.K. Allen, and F. Mistree, 'Metrics for Assessing Design Freedom and Information Certainty in the Early Stages of Design', *Journal of Mechanical Design*, **120**, 628–635, (dec 1998). Transactions of the ASME.

[22] M. Veron, H. Fargier, and M. Aldanondo, 'From CSP to Configuration Problems', in *AAAI-99 Workshop on Configuration*, Orlando, Florida, (jul 1999).