

# High-Level Modeling of Component-based CSPs

Raphaël Chenouard<sup>1</sup>, Laurent Granvilliers<sup>2</sup>, and Ricardo Soto<sup>3</sup>

<sup>1</sup> IRCCYN, Ecole Centrale de Nantes, France

<sup>2</sup> LINA, CNRS, Université de Nantes, France

<sup>3</sup> Escuela de Ingeniería Informática

Pontificia Universidad Católica de Valparaíso, Chile

{ricardo.soto, laurent.granvilliers}@univ-nantes.fr

{raphael.chenouard}@irc cyn.ec-nantes.fr

**Abstract.** Most of modern constraint modeling languages combine rich constraint languages with mathematical notations to tackle combinatorial optimization problems. Our purpose is to introduce new component-oriented language constructs to manipulate hierarchical problems, for instance for modeling engineering system architectures with conditional sub-problems. To this end, an object-oriented modeling language is associated with a powerful constraint language. It offers the possibility of defining conditional components to be activated at solving time, declaring polymorphic components whose concrete types have to be determined, and overriding model elements. We illustrate the benefits of this new approach in the modeling process of a difficult embodiment design problem having several architectural alternatives.

## 1 Introduction

Constraint programming (CP) is a generic framework allowing users to state various kinds of constraint satisfaction problems (CSPs). Several specialized paradigms emerged to tackle specific concepts, for instance conditional CSPs [5] and composite CSPs [13].

A CSP is conditional when its sub-problems may not participate in the solutions. The activation of sub-problems is subjected to constraints to be verified during the solving process. A composite CSP can be seen as a hierarchy of sub-problems. It captures the inherent structure of a component-based system. Combining both features, composite and conditional CSPs make it possible to formulate models of complex systems and to study some variations in their architectures. This class of problems finds many applications in the configuration and design areas [12, 5, 8].

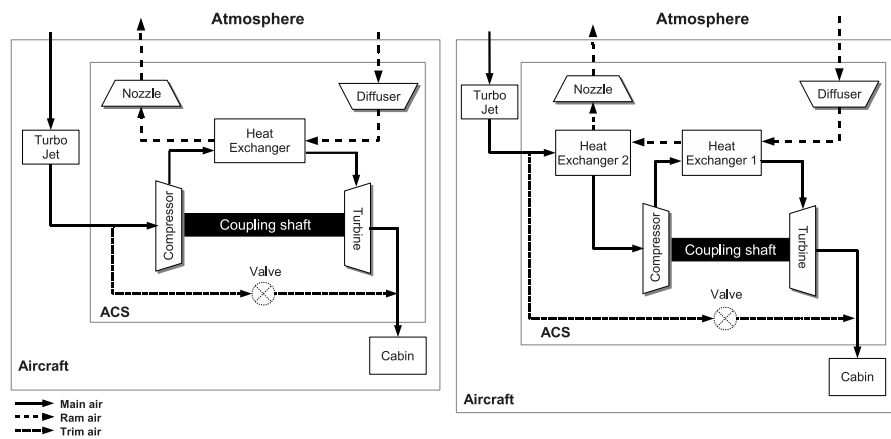
Modeling such problems is not trivial, as it is necessary to handle various concerns e.g., the component hierarchy, the activation of sub-problems, and even the definition of families of components sharing some properties. In this paper, we present a new approach to elegantly represent component-based CSPs. We extend the s-COMMA modeling language [15] by adding a new set of language constructs. We firstly introduce the concept of conditional (optional) component and we provide a simple construct to employ them in constraint models.

Depending on constraint-based conditions, it is possible to dynamically modify the problem architectures by activating new components. Second, components can be polymorphic. It is possible to define families of components, whose types can be refined at solving time. Third, as usual in object-oriented languages, model elements can be overridden in sub-types to facilitate the implementation of sub-classes. Finally, the language also makes it possible to handle components off the shelf or catalogs by means of compatibility constraints, which are sets of tuples to be assigned to a given set of variables.

Modeling complex real-world CSPs is a really hard task and s-COMMA is the result of several years of experience tackling that kind of problems. Now, let us illustrate such new modeling features by presenting a real-world CSP from Dassault Aviation about the design of an air conditioning system (ACS) for aircrafts.

### 1.1 Example

The ACS in an aircraft roughly mixes hot air from the turbojet (main air) and cold air from atmosphere (ram air) to inject air at right temperature and pressure in the cabin (see [3] for more details). Two topologies connecting several components are depicted in Fig. 1. On the left, only one heat exchanger is used to transfer the calorific energy between the main air and the ram air. On the right, a second heat exchanger is used to pre-cool the main air. Several other topological changes may also be considered in a more complete analysis of this complex system.



**Fig. 1.** Two possible topologies of an air conditioning system in an aircraft.

These architectures are hierarchical, since every component can be refined as a complex system. Moreover, the choice of topology may depend on conditions. In this problem, the goals may be to simultaneously study:

- the embodiment design of the ACS, which in particular requires the solving of nonlinear constraints over continuous variables related to the geometry or the physical behavior of the system,
- the feasibility of each architectural design alternative.

Thus, we need a high-level modeling language being able to represent at least components subject to constraints, discrete and continuous variables, and conditional components.

## 1.2 Outline

The remaining of this paper is organized as follows. s-COMMA and the new modeling features are presented in Section 2. The architecture supporting the modeling language is shortly described in Sect. 3. The related work and conclusions follow.

## 2 Modeling

The core of the s-COMMA modeling language is briefly described now. A model is a collection of classes to be linked through usual composition and inheritance relations. The main class corresponds to the model entry, as follows.

```

1 // Model file
2 main class Aircraft {
3   ACS acs; // the ACS
4   TurboJet tj; // the turbojet
5
6   constraint Airflows {
7     tj.TIn = TRam; // tj input = atmosphere
8     tj.TOut = acs.THotIn; // tj output = ACS input
9     acs.THotOut = TCab; // ACS output = cabin
10    acs.TColdIn = TRam; // ACS input = atmosphere
11    ...
12  }
13 }
14
15 class ACS {
16   HeatExchanger ex1; // the (first) heat exchanger
17   Turbine turbine; // the turbine
18
19   real THotIn in [200,1000]; // temperature in K of input hot air
20   real TColdIn in [200,1000]; // " in K of input cold air
21   real THotOut in [200,1000]; // " in K of output hot air
22   real TColdOut in [200,1000]; // " in K of output cold air
23   real epsilon in [0,1]; // heat transfer efficiency
24   ...
25 }
26
27 class HeatExchanger {
28   real THotIn in [200,1000]; // temperature in K of input hot air
29   real TColdIn in [200,1000]; // " in K of input cold air
30   real THotOut in [200,1000]; // " in K of output hot air
31   real TColdOut in [200,1000]; // " in K of output cold air
32   ...

```

In this model, the aircraft is composed of an air conditioning system `acs` and a turbojet `tj`, being instances of other classes. A constraint block `Airflows`

is defined to link the temperatures of air flows between the atmosphere, the turbojet, the ACS, and the cabin. A set of equality constraints is simply stated between air temperature attributes of `acs` and `tj`, and the flight conditions, namely the atmosphere air temperature `TRam` and the cabin air temperature `TCab`. These two parameters can be defined as constants in a data file for given flight conditions, as follows:

```

1 // Data file
2 real TRam := 220.055; // atmosphere air temperature in K at 10500m
3 real TCab := 280; // expected cabin air temperature in K

```

The `ex1` attribute corresponds to the heat exchanger in the first topology, and to the first one in the second topology. That means that every topology at least requires one exchanger. The presence of one or two exchangers will be more precisely discussed in Sect. 2.2.

For the sake of clarity, only a small extract of the full model is presented above. The other components are defined in the same way. In the rest of the section we focus on the new component-oriented language constructs.

## 2.1 Catalogs

In most configuration and design problems, it is necessary to reuse some components from the shelf, given as catalogs stating tuples of values for their main characteristics. For instance, a heat exchanger is made of exchange surfaces to be chosen from a set of known shapes. This can simply be modeled by means of a Boolean formula that restricts the possible values of several variables such as `rh`, `bh`, and `betah`.

```

1 class HeatExchanger {
2   ...
3   // Variables related to the exchange surfaces
4   real bh in [0,0.1];
5   real rh in [0,0.1];
6   real betah in [100,1.0e4];
7   ...
8   constraint SurfacesCatalog {
9     (rh=7.7089e-4 and bh=6.35e-3 and betah=1.204e+3) or
10    (rh=3.61315e-3 and bh=1.905e-3 and betah=2.496e+2) or
11    (rh=6.6167e-4 and bh=1.051e-2 and betah=1.368e+3) or
12    (rh=6.6992e-4 and bh=9.525e-3 and betah= 1.25e+3);
13  }

```

This is a natural representation of catalogs, but it is not suitable to deal with a higher number of constraints, as the model becomes confuse. To avoid this, it is possible to compact the model by using a compatibility constraint.

```

1   constraint SurfacesCatalog {
2     compatibility
3     (rh, bh, betah) {
4       (7.7089e-4, 6.35e-3, 1.204e+3);
5       (3.61315e-3, 1.905e-2, 2.496e+2);
6       (6.6167e-4, 1.051e-2, 1.368e+3);
7       (6.6992e-4, 9.525e-3, 1.25e+3);
8     }
9   }

```

## 2.2 Conditional components

In various CP problems, it is necessary to model elements whose activation depends on the satisfaction of some conditions. For instance, consider the heat exchanger whose properties vary depending on the temperature of the input hot air (`THotIn`). This is commonly modeled by adding a set of implications that activate the involved constraints.

```

1 // Within the ACS class
2 constraint AirflowsOne {
3   (THotIn <= 600) -> ex1.TColdOut = nozzle.TIn;
4   (THotIn <= 600) -> THotIn = compressor.TIn;
5 }
6 constraint HeatEfficiencyOne {
7   (THotIn <= 600) -> epsilon = ex1.epsilon;
8 }
9 }
```

The model becomes more complicated if we need to model the activation of a new object depending on a condition. For instance, in the second ACS topology, an additional heat exchanger is required when the input hot temperature exceeds 600 K. One manner to state that is to include the second heat exchanger within the class and to activate the corresponding constraints.

```

1 // Within the ACS class
2 HeatExchanger ex2; // the second exchanger
3
4 constraint AirflowsTwo {
5   (THotIn > 600) -> ex1.TColdOut = ex2.TColdIn;
6   (THotIn > 600) -> ex2.TColdOut = nozzle.TIn;
7   (THotIn > 600) -> THotIn = ex2.THotIn;
8   (THotIn > 600) -> ex2.THotOut = compressor.TIn;
9 }
10 constraint HeatEfficiencyTwo {
11   (THotIn > 600) -> epsilon = (ex1.epsilon + ex2.epsilon)/2;
12 }
13 }
```

Additionally, it is necessary to deactivate the object attributes (when the condition is not satisfied) in order to avoid useless splitting operations during the search. This can roughly be done by assigning to the variables its minimum domain value. Unfortunately, the resultant model is too verbose and hard to understand.

```

1 constraint deactivateAttributes {
2   not(THotIn > 600) -> (ex2.THotIn = minDom(ex2.THotIn));
3   not(THotIn > 600) -> (ex2.TColdIn = minDom(ex2.TColdIn));
4   not(THotIn > 600) -> (ex2.THotOut = minDom(ex2.THotOut));
5   not(THotIn > 600) -> ...
6 }
```

To make the definition of such a conditional formulation more concise and understandable, we introduce a new conditional statement. This new construct considers a name, a condition, and a sequence of elements to activate if the condition is satisfied. For instance, the constraints defined within the `AirflowsOne` constraint block are only activated if the `OneExchanger` conditional block evaluates to true.

```

1 // Within the ACS class
2 cond OneExchanger (THotIn <= 600) {
```

```

3   constraint AirflowsOne {
4       ex1.TColdOut = nozzle.TIn;
5       THotIn = compressor.TIn;
6   }
7   constraint HeatEfficiencyOne {
8       epsilon = ex1.epsilon;
9   }
10  }

```

The integration of conditional objects is performed in the same way. This feature is more powerful since the set of variables and constraints embedded in the object can be activated at once, as for instance in the second ACS topology. The second heat exchanger is activated only if the `THotIn` attribute exceeds 600 K. It is also possible to include additional constraints acting over the conditional object, for example to establish the new links between air flows, and to compute the new heat efficiency.

```

1  // Within the ACS class
2  cond TwoExchangers (THotIn > 600) {
3      HeatExchanger ex2; // the second exchanger
4
5      constraint AirflowsTwo {
6          ex1.TColdOut = ex2.TColdIn;
7          ex2.TColdOut = nozzle.TIn;
8          THotIn = ex2.THotIn;
9          ex2.THotOut = compressor.TIn;
10     }
11     constraint HeatEfficiencyTwo {
12         epsilon = (ex1.epsilon + ex2.epsilon)/2;
13     }
14 }

```

Note that the conditional block can also be located in a higher class (considering the composition's hierarchy), specifying that they apply on a given object. For instance, now the conditional block is stated within the `Aircraft` class, but it remains acting over the `acs` object.

```

1  class Aircraft {
2      ...
3      cond TwoExchangers (THotIn > 600) on acs {
4          ...

```

The main difference here is to modify a single object rather than the class itself. That allows one to manipulate instances of a same class with few modifications.

### 2.3 Polymorphic components

Abstract or non final classes in object-oriented programming languages allow one to uniformly manipulate instances of sub-classes. From a modeling viewpoint, it is common to specify sub-types corresponding to families of systems differing in some aspects. Our goal is to give users a mean for declaring some objects of a given type and to let the solving engine dynamically refine this type, i.e., to allow non deterministic choices of component types.

We just define the new `polymorphic` keyword to be used in the declaration of a variable. As a consequence, every polymorphic object of type `T` occurring in a solution may have type `T` or a sub-type of `T`. In this case, we impose that the full type list from the inheritance hierarchy has to be explored.

This feature is illustrated by considering three main families of heat exchangers depending on directions of the cold and hot air flows: co-current flows, counter-current flows, and cross-current flows as depicted in Fig. 2. Each type from this family has its own way to compute the exchange of heat energy between the two air flows.

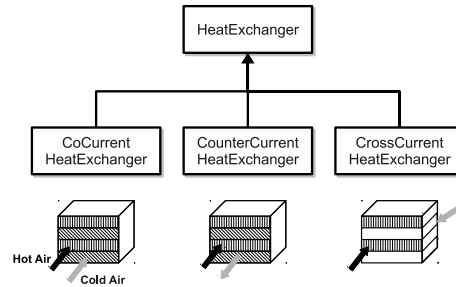


Fig. 2. A simple inheritance hierarchy for heat exchangers.

The family of exchangers can be naturally represented by an inheritance graph as shown in Fig. 2. It directly follows the following classes.

```

1 class HeatExchanger { ... }
2 class COCHExchanger extends HeatExchanger { ... }
3 class CTCHExchanger extends HeatExchanger { ... }
4 class CRCHExchanger extends HeatExchanger { ... }

```

Now let the (first) heat exchanger from the ACS to be polymorphic. Furthermore, suppose that the second heat exchanger must be of type `CRCHExchanger` and that, in this case, the first exchanger must be of the same type. The following piece of model results.

```

1 // Within the ACS class
2 polymorphic HeatExchanger ex1; // polymorphic instance
3
4 constraint Airflows { // shared constraint between
5   ex1.THotOut = turbine.TIn; // the two topologies
6   ex1.TColdIn = diffuser.TOut;
7   ...
8 }
9
10 cond TwoExchangers (THotIn > 600) {
11   CRCHExchanger HeatExchanger ex2; // the second exchanger
12
13   constraint TypeExchangerOne { // type restriction
14     typeOf(ex1) = typeOf(ex2); // for the first exchanger
15   }
16   ...
17 }

```

Introducing polymorphic instances leads to the need for manipulating instance types. In the example, the type of the first exchanger is restricted using the `typeOf` keyword (lines 13 to 15). That may be compared with runtime type checking mechanisms in computer programming languages, such as `instanceof` in Java or `type` in Python. The main difference is the declarative nature of the primitives in our language, which are just constraints on types.

## 2.4 Overriding elements

The purpose of overriding mechanisms is to allow a sub-type from a hierarchy to provide a specific definition of an element — variable, constraint block, conditional block— defined in some ascending type. To this end, it suffices to declare an element using a naming correspondence.

In the ACS, the areas of the exchange surfaces are defined in the `HeatExchanger` base type. They are valid for all the given sub-types, except for the cold exchange surface of `CRCHExchanger` heat exchangers. In this case, it suffices to override the `ColdExchangeSurfaceArea` constraint, as follows.

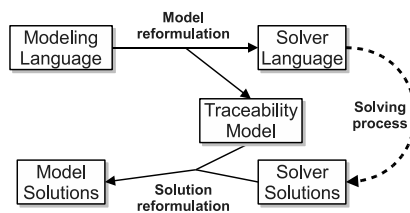
```

1  class HeatExchanger {
2    // let Lx, Ly, and Lz be the dimensions
3    // let Afh and Afc be the related areas
4
5    constraint HotExchangeSurfaceArea {
6      Afh = Lx*Lz;
7    }
8    constraint ColdExchangeSurfaceArea {
9      Afc = Lx*Lz;
10   }
11 }
12
13 class CRCHExchanger extends HeatExchanger {
14   constraint ColdExchangeSurfaceArea {
15     Afc = Ly*Lz; // constraint overriding
16   }
17 }

```

## 3 Architecture

s-COMMA has been implemented on a model-driven solver-independent platform. Such an architecture allows the automatic reformulation of a s-COMMA model into different executable solver models. A model-driven engineering approach has been implemented to perform the necessary transformations among source, intermediate, and target models. Although, those transformations are not detailed here for space reasons, an extended presentation about them can be found in [15, 1, 2].



**Fig. 3.** Reformulation process.

A general scheme of the reformulation process is depicted in Fig. 3. An s-COMMA model is the input of the system. This model is mapped to an intermediate model in order to reformulate the language constructs not supported

at the solver level. Those transformations are standard and are based on the previous implementations of `s-COMMA`. For instance, objects are flattened i.e., every variable and constraint are just incorporated into the intermediate model. Compatibility constraints are mapped to the corresponding set of Boolean expressions. Conditional blocks are transformed into a set of implications, and polymorphic elements are stated as single conditional blocks. These conditional blocks are then treated as logical formulas. Once the reformulation is finished, a solver model is generated from the intermediate one, which is launched to obtain the results. Finally, the solution set is analyzed according to a traceability model to display the solutions in a correct object-oriented format.

## 4 Related Work

Zinc [9], MiniZinc [11] (successors of OPL [16]), and Essence [4] are the state-of-the-art constraint modeling languages. They provide a rich semantics for modeling different kinds of problems. In particular, Zinc offers the possibility of tackling specific applications domains by means of user-defined predicates. However, a main problem is that no object-oriented features are provided. Hence, it is difficult to naturally capture the structure of component-based problems.

Compatibility constraints can undoubtedly be represented by these languages as Boolean formulas, and conditional blocks can be simulated by using Boolean variables and/or implications as illustrated in Sect. 2. But, the result seems to be a hack of the model rather than a natural problem formulation. Finally, there is no mechanism to manage families of components through polymorphism.

COB [7] is an object-oriented constraint modeling language that can capture hierarchical problem structures. It provides a common language to post constraint models, which can be enriched with Prolog-like predicates. But, once again it lacks of compatibility constraints, conditional statements, and polymorphism. Additionally, less relevant to this paper, but not less important, the underlying architecture of COB is solver-dependent, being not possible to launch a model in different solving engines.

## 5 Conclusion

In this paper, we have presented new language constructs for modeling component-based CSPs. These constructs allow one to define more concise and understandable models, that naturally represents complex hierarchical problems. For instance, compatibility constraints avoids the definition of confuse Boolean expressions. Conditional blocks are a mean to express variables, constraints, and even objects whose relevance on the model depends on a given condition. Polymorphism is suitable for modeling families of components sharing some properties. Such constructs have been integrated on the `s-COMMA` language with the spirit of designing a full component-oriented language, an important CP challenge as stated in [8].

We believe that an important research direction is about solving strategies for component-based CSPs, interesting studies are pointed out in [6, 5, 14, 10]. The main problem is that the set of required techniques may not be present in any existing solver, such as consistency techniques for mixed variables, interval methods for continuous problems, various search strategies, decomposition algorithms, and so on. Developing a new solver is a really hard task, and the solution may be to make several tools cooperate inside a CP platform.

## References

1. R. Chenouard, L. Granvilliers, and R. Soto. Model-Driven Constraint Programming. In *Proceedings of ACM SIGPLAN PPDP*, pages 236–246. ACM Press, 2008.
2. R. Chenouard, L. Granvilliers, and R. Soto. Rewriting Constraint Models with Metamodels. In *Proceedings of SARA*, pages 42–49. AAAI Press, 2009.
3. R. Chenouard, P. Sébastien, and L. Granvilliers. Solving an Air Conditioning System Problem in an Embodiment Design Context Using Constraint Satisfaction Techniques. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 18–32, 2007.
4. A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez Hernández, and I. Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.
5. E. Gelle and B. Faltings. Solving Mixed and Conditional Constraint Satisfaction Problems. *Constraints*, 8:107–141, 2003.
6. F. Geller and M. Veksler. Assumption-Based Pruning in Conditional CSP. In *Proceedings of CP*, volume 3709 of *LNCS*, pages 241–255. Springer, 2005.
7. B. Jayaraman and P. Tambay. Modeling Engineering Structures with Constrained Objects. In *Proceedings of PADL*, volume 2257 of *LNCS*, pages 28–46. Springer, 2002.
8. U. Junker. *Handbook of Constraint Programming*, chapter Configuration, pages 837–873. Elsevier, 2006.
9. K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia de la Banda, and M. Wallace. The Design of the Zinc Modelling Language. *Constraints*, 13(3):229–267, 2008.
10. M. Mouhoub and A. Sukpan. Managing Conditional and Composite CSPs. In *Proceedings of Canadian Conference on AI*, volume 4509 of *LNCS*, pages 216–227. Springer, 2007.
11. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Proceedings of CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
12. B.A. O’Sullivan. *Constraint-Aided Conceptual Design*. Professional Engineering Publishing, 2001.
13. D. Sabin and E. C. Freuder. Configuration as Composite Constraint Satisfaction. In *Proceedings of AI and MRP Workshop*, pages 153–161. AAAI Press, 1996.
14. M. Sabin, E. C. Freuder, and R. J. Wallace. Greater Efficiency for Conditional Constraint Satisfaction. In *Proceedings of CP*, volume 2833 of *LNCS*, pages 649–663. Springer, 2003.
15. R. Soto and L. Granvilliers. The Design of COMMA: An Extensible Framework for Mapping Constrained Objects to Native Solver Models. In *Proceedings of ICTAI*, pages 243–250. IEEE Computer Society, 2007.
16. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.