Computational Design Synthesis Using Model-Driven Engineering and Constraint Programming

Raphael Chenouard[1(✉)], Chris Hartmann[1,2], Alain Bernard[1], and Emmanuel Mermoz[2]

[1] Ecole Centrale de Nantes, IRCCyN UMR CNRS 6597, 1 Rue de la No BP 92101, 44321 Nantes Cedex 3, France
{raphael.chenouard,alain.bernard}@irccyn.ec-nantes.fr, chris.hartmann@airbus.com

[2] Airbus Helicopters, Aéroport de Marseille Provence, 13700 Marignane, France
emmanuel.mermoz@airbus.com

Abstract. This paper introduces a new process for computational design synthesis. It starts from functional requirements to generate one or more topologies of components. This process is implemented using Model-Driven Engineering techniques and Constraint Programming solving capabilities. Model transformations are used to transform functions and available components to a CSP. This problem is solved with a CSP solver, which solutions are transformed to topological architectures. The process is successfully applied on the design synthesis of an autonomous generator. It produces about 60 relevant solutions from which we found some existing product architectures.

1 Introduction

Design synthesis is a hard task in the design process of a product. It is one step within the preliminary design phase of a system of interest, when considering a common design process [10]. The design synthesis task ends-up with a set of architectures related to a functional decomposition derived from the stakeholders' needs. The modeling of products during preliminary design phases is generally based on three aspects: function, behavior and structure [4].

Some previous works in design synthesis are based on graph grammars and rules to build graphs corresponding to relevant topologies of components [6]. These rules are based on well-known principles of solutions or functional decomposition in a given context. The major advantages of this kind of approaches is that it generates solutions that fit good practices and designer experience. However, the search space of possible solutions may be only partially explored. Thus, some innovative and efficient solutions may not be found.

Some recent works mainly in the field of embedded systems investigate this kind of problem as Design Space Exploration (DSE) [8,16]. These works use as well model-driven engineering and optimization techniques to automate the

© Springer International Publishing AG 2016
P. Milazzo et al. (Eds.): STAF 2016, LNCS 9946, pp. 265–273, 2016.
DOI: 10.1007/978-3-319-50230-4_20

definition of a valid solution. [8] like [6] uses graph theory to define rules that guide the building of solutions. Some cut-off criteria are used to improve the exploration procedure which defines new states by applying graph transformation rules using a selection heuristic. In [16], the aim is to define a more generic framework being able to address various kinds of DSE like resource allocation problems, routing problems or configuration problems. One of the major benefits of this approach is its wide application range and its solver independence. However, the designer has to define a metamodel template used for the exploration. Obviously it makes the exploration easier, but it requires to know in advance the main structure of valid solutions even if their language propose mechanisms to deal with alternatives or optional elements.

Another previous approach also mixed Model-Driven Engineering (MDE) and Constraint Programming (CP) [2] to define a framework for modeling problems independently from a solver like in the Model-Driven Architecture philosophy, but applied here to mathematical problem modeling and solving. This work was also followed by the definition of high level modeling concepts to ease the definition of Constraint Satisfaction Problems (CSPs) in order to represent design problems [3]. The behavioral aspect of a product was the main target, whereas functional and structural concerns were not really investigated.

In this paper, we propose a method based on MDE and CP to compute product topologies from a brief functional description. The generation of topologies must only satisfy the functional requirements. We do not take into account, at this step, the behavioral aspect of a product and we restrict the structure definition to a topological architecture: a set of inter-connected components. Our aim is to explore all possible solutions and provide more innovative architectures that may not be found using classical design processes. Moreover, we do not want to use pre-defined rules or patterns that will always lead to the same kinds of solution principles and exclude possible promising solutions.

The next section introduces the main process and modeling elements regarding designers activities for design synthesis. Section 3 deals with automated solving of a computational design synthesis problem. Section 4 presents an application of the method on a concrete case with a discussion of the proposed process issues, before ending with a conclusion and the future works description.

2 Design Synthesis Automated Process

As said previously, design synthesis aims at generating a product architecture from needs and requirements [1]. In this paper, we focus on the transformation of functional needs to topological architectures, namely networks of components. In most existing work, designers first decompose functions to define a functional architecture [6], then allocate functions to physical components and check feasibility and performances [17].

We propose in this paper to directly compute a topological architecture without investigating too deeply the functional architecture. Our aim is to maximize innovative solutions without using classical design patterns that will always produce the same solution principles. Thus, we just want to use high-level functions

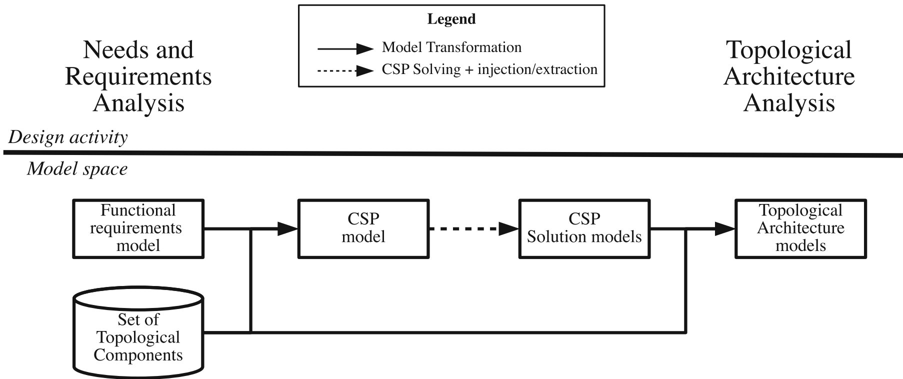

Fig. 1. Main process of the proposed method.

- issuing from stakeholders needs - and a database of allowed topological components (see Fig. 1). These high-level functions define the functional requirements from which we compute satisfying architectures. The following subsections presents the two main metamodels we used as input and output of our automated process implemented in the Eclipse environment with ATL transformation Language [11]. The solving is done using CSP formalism with classical CP solving methods [12]. Computed solutions are transformed into topological architectures and finally designers can analyse and investigate the best one(s) for physical feasibility analysis.

2.1 Functional Requirements Modeling

Since more than a decade, researchers investigate the best manner to represent functions within a design process. The kind of words or verbs to use is out of the scope of this paper. We refer to this previous work [9] to deal with this issue. We want to define the main concepts that are used in the proposed transformation process to state input models like in [7].

Then, a function is defined by a name (that should be an action verb) and a set of flows (oriented or not). Flows can be of three main categories: material, energy and signal. Functions and flow may have some properties defined by a name and a unit (e.g. an electrical flow is often defined with 2 properties: *current* with unit A and *voltage* with unit V). A model of functional requirements is simply a set of functions and flows instances as shown in Fig. 2.

2.2 Topological Architecture Modeling

We consider that a topological architecture is a network of components, namely a set of inter-connected components. This definition is similar to the definition in the systems engineering domain [14]. Then, a component is mainly defined by its name and its interfaces relating to flows (see Fig. 3). Components are

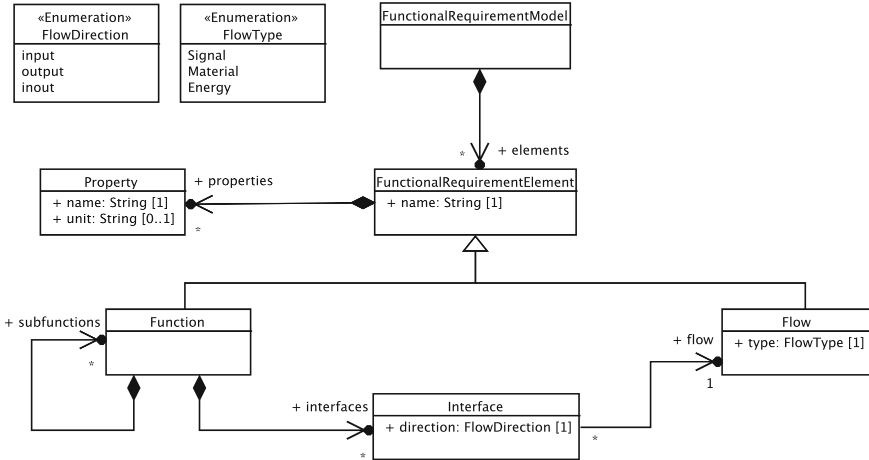

Fig. 2. Metamodel for simplified functional requirements definition.

connected through their interfaces which must be of compatible flows. Some concepts are similar to those defined in the functional requirements metamodel, like flow and property. Components are close to functions, but the function concept relates to main functions, whereas components may integrate additional flows corresponding to induced effects and they are connected to form a network. Moreover, they are defined as generic abstractions of real components like for instance a generic piston engine or an electrical battery.

Components are not considered as composite since we only consider atomic ones. We focus on their connections within an architecture at a given level of decomposition. Following a systemic approach, one can easily define functional requirements for a component and apply recursively this process to define its composition. In fact, we consider that a component only refers to a physical element that interacts with other elements of the same decomposition level. In this way, several components of the same type are just considered as different components with same properties and interfaces definition. For instance, we may have several piston engines with identical characteristics (maximum power, efficiency curve, input/output interfaces), but we consider them as different components in a topological architecture.

3 Solving a Design Synthesis Problem

Passing from functional requirements to a physical architecture is not obvious and cannot be processed using a simple model transformation. We propose in this paper to formulate a graph problem that can be solved using CP solvers. The objective is to find a set of connected nodes (i.e. used component interfaces) and a set of isolated ones (i.e. unused component interfaces) with respect to a set of constraints related to the possible connections.

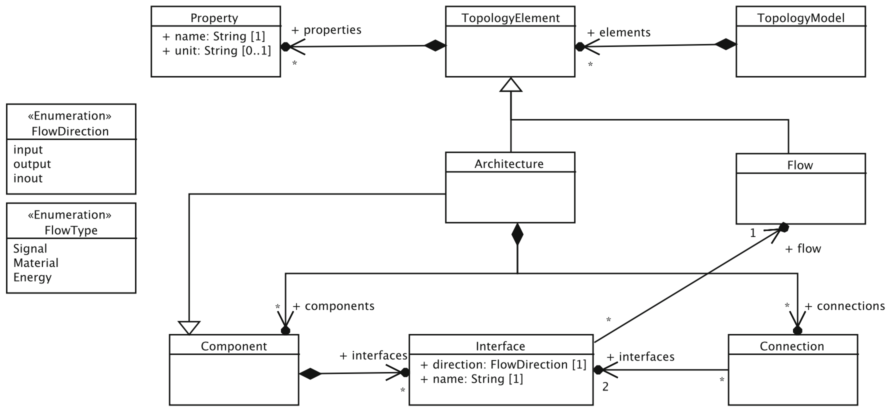

Fig. 3. Metamodel for topological architecture definition.

Given a database of allowed components and the set of functions to satisfy, we generate a mathematical problem which solutions are connected graphs compatible with the following constraints:

1. connections can only exist between compatible interfaces,
2. function flows must be satisfied by input/output interfaces of the whole system,
3. all interfaces of a component must be connected or none of them.

In our CSP, decision variables are the connections between interfaces (components and functional requirements). Thus, we use a matrix of binary variables to represent these decision variables. We can easily pre-compute the compatible and incompatible interfaces of each given interface using its flow description to eliminate some trivial decision variables. Concretely, we compute a matrix of binary values defining allowed and not allowed connections and we set constraints fixing these decision variables if the connection is not allowed. Thus, only the second and third set of constraints are used to restrict the domain of variables during the solving process [15].

We use a simplified metamodel to define CSPs as shown in Fig. 4. A CSP is defined as three sets: domains, variables and constraints. Since we only use integer and binary variables, no additional domain kinds are considered. Constraints are not detailed here, but consist of classical logical and arithmetical expressions [2]. Since we use MiniZinc concrete syntax, we take advantage of some additional high-level constructs like matrices of variables or parameters, forall and if-else constraints or sum function calls [13]. The solving is carried out with the default solver of MiniZinc 2.0.12 distribution.

After the solving phase, we get a set of solutions. A solution is simply a list of couples (value, variable) for which all constraints are satisfied (see Fig. 5). Obviously all variables must have a value to get a complete solution. Since decision variables are connections between interfaces of components, it is easy to identify

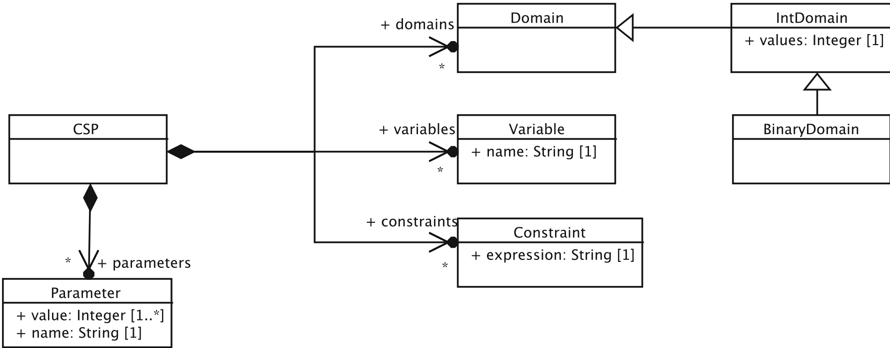

Fig. 4. Simplified metamodel for CSP modeling.

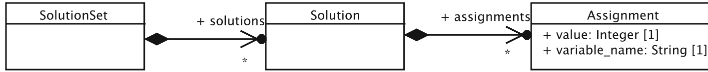

Fig. 5. Metamodel for solution modeling.

which ones are used and how they are linked to each others. A last transformation step is used to generate a topology from the set of used components and a CSP solution.

One drawback of this modeling, is the possible huge number of variables. For n interfaces (from a given set of allowed components) and m interface from input and output functions to satisfy, we have more than $n^2 + m * n$ variables. Nevertheless, we use binary variables and the scaling of CP solving algorithms stays satisfactory.

4 Application and Discussion

We applied our approach to an autonomous generator design synthesis problem. We consider three high-level functions:

- the system must start/stop on demand,
- the system must produce electrical energy,
- the system must follow a voltage order (e.g. between 110 V, 220 V and 370 V),

These three functions imply two input flows: (1) the voltage order and (2) an on/off signal; and only one output flow: electrical energy.

We use a set of 10 allowed components corresponding to 39 interfaces. So, we get $1521 + 117$ binary variables. These components include the environment as a source for air and a sink for (exhaust) gas and thermal energy.

We obtain about 60 solutions. Figure 6 shows one solution that uses all allowed components. A turbine and a piston engine are used to produce the

mechanical energy for an alternator which produces electrical energy. An electrical engine is used, since it is required to start the turbine and the piston engine. It produces mechanical energy and it receives the start signal and electrical energy. No electrical energy flow is defined in the input functional requirements, so a battery is used to feed the electrical engine. This battery can also be used to store and deliver the produced electricity. It also may improve the electricity quality as the battery can soften the power demand.

For each solution and the corresponding selected component descriptions, an architectural topology is generated. For printing purposes, we also generate a DOT model processed with the GraphViz compiler [5] as it can be seen on Fig. 6.

We apply our process to an autonomous generator problem. We only consider 10 topological components, but we get more than 60 topologies. All these

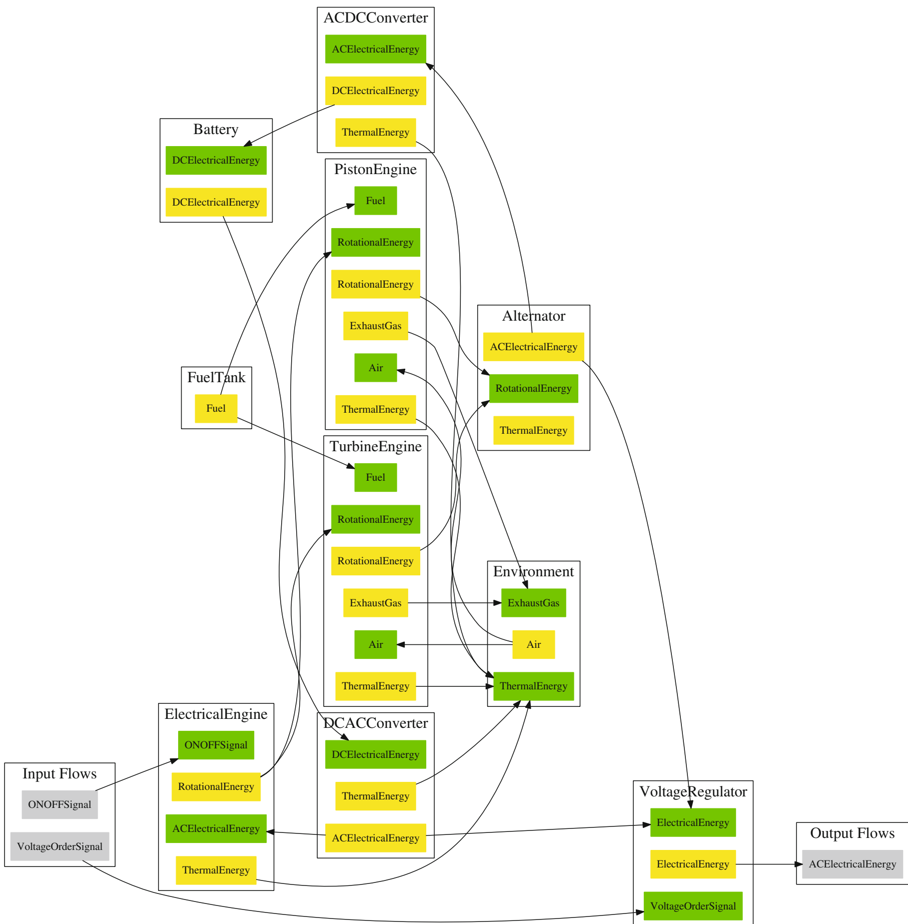

Fig. 6. Example of a solution obtained by the solving process.

solutions are valid in terms of flow connections and we were able to find some architectures used in existing products.

On this example, we do not use several occurrences of components. Some other experiments show that many similar (i.e. symmetrical) solutions are computed and we can expect an exponential rise of the solution number according to the number of allowed components (and their number of interfaces). The next step for the designer is to check the physical feasibility of a topological architecture. Working with so much solutions is not realistic on bigger design synthesis problems even if we can automate many steps, but additional constraints can be easily integrated in our approach to take into account other requirements or performance criteria.

5 Conclusion and Future Work

In this paper, we present an innovative process to automate the design synthesis of system architectures. We implement it using MDE tools and CP techniques to compute relevant topologies. The designer has just to define functional requirements and select candidate components, then the automated process will produce all possible topologies. We define several simplified metamodels for the function definitions, the CSP model, the CSP solution, the topological architecture model and we use an existing DOT language metamodel for printing the computed topologies. We apply this process on a real example using a set of allowed components. It proves the relevance of the process, even if some improvement must be done.

Indeed, we have to consolidate our transformations and we have to automate the whole process, since some steps are manually launched and achieved (e.g. some model injections). Nevertheless, we expect to link our process with existing system modeling languages like SysML. Functional requirements may automatically be extracted. Topological architectures can also be defined using block definition diagrams and internal block diagrams.

Several harder issues must be investigated after this work. The major one is the number of solutions and the symmetries appearing with multiple occurrences. Without a drastic reduction of the number of computed solutions, the process will not be fully usable for real-world design synthesis problems. In CP, some existing work deals with symmetry breaking techniques [18] and we hope to reduce drastically the number of computed solutions.

Another way to reduce valid topologies is to add more constraints about the design synthesis problem. We only use high-level functional requirements, but additional knowledge may be integrated as constraints in our CSPs to assess the feasibility in terms of physical behavior. However, these constraints may often lead to nonlinear constraints. In this case, we have to deal with Mixed-Integer NonLinear Problems (MINLPs) which are harder to solve than current Integer Linear Problem (ILP). We can also use an optimization algorithm to reduce the number of computed solutions, but we have to define generic metrics or performance criteria related to the topological aspects of the computed solutions.

Some additional knowledge may also be integrated to deal with performance criteria coming from needs and requirements.

References

1. Cagan, J., Campbell, M.I., Finger, S., Tomiyama, T.: A framework for computational design synthesis: model and applications. ASME. J. Comput. Inf. Sci. Eng. **5**(3), 171–181 (2005)
2. Chenouard, R., Granvilliers, L., Soto, R.: Model-driven constraint programming. In: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), pp. 236–246 (2008)
3. Chenouard, R., Granvilliers, L., Soto, R.: High-level modeling of component-based CSPs. In: Rocha Costa, A.C., Vicari, R.M., Tonidandel, F. (eds.) SBIA 2010. LNCS (LNAI), vol. 6404, pp. 233–242. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16138-4_24
4. Gero, J.S.: Design prototypes: a knowledge representation schema for design. AI Mag. **11**(4), 26 (1990)
5. Graphviz: Graph visualization software. http://www.graphviz.org
6. Helms, B., Shea, K.: Computational synthesis of product architectures based on object-oriented graph grammars. J. Mech. Des. **134**(2), 1–14 (2012)
7. Hartmann, C., Chenouard, R., Mermoz, E., Bernard, A.: Formulation of a design problem for computational pre-design. In: Virtual Concept Workshop (2016)
8. Hegeds, A., Horvth, A., Varr, D.: A model-driven framework for guided space exploration. Autom. Softw. Eng. **22**(3), 399–436 (2015)
9. Hirtz, J., Stone, R.B., McAdams, D.A., Szykman, S., Wood, K.L.: A functional basis for engineering design: reconciling and evolving previous efforts. Res. Eng. Des. **13**(2), 6582 (2002)
10. Pahl, G., Beitz, W.: Engineering Design: A Systematic Approach. Springer, London (1995)
11. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(12), 3139 (2008)
12. Kumar, V.: Algorithms for constraint satisfaction problems: a survey. AI Mag. **13**(1), 32–44 (1992)
13. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74970-7_38
14. Rechtin, E.: Systems Architecting: Creating and Building Complex Systems. Prentice Hall, Englewood Cliffs (1991)
15. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier, New York (2006)
16. Saxena, T., Karsai, K.: Towards a generic design space exploration framework. In: IEEE 10th International Conference on Computer and Information Technology (CIT), pp. 1940–1947 (2010)
17. Umeda, Y., Tomiyama, T., Yoshikawa, H.: FBS modeling: modeling scheme of function for conceptual design. In: Proceedings of the 9th International Workshop on Qualitative Reasoning (1995)
18. Walsh, T.: General symmetry breaking constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 650–664. Springer, Heidelberg (2006). doi:10.1007/11889205_46