HAL
open science

Model-based trade studies in systems architectures design phases

Nicolas Albarello

To cite this version:

Nicolas Albarello. Model-based trade studies in systems architectures design phases. Other. Ecole Centrale Paris, 2012. English. NNT : 2012ECAP0052 . tel-00879858

HAL Id: tel-00879858

https://theses.hal.science/tel-00879858

Submitted on 5 Nov 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EADS INNOVATION WORKS

CENTRALE PARIS

Model-based trade studies in systems architectures design phases

by
Nicolas Albarello

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Ecole Centrale Paris - France

December 2012

Thesis director :	Dr. Jean-Claude Bocquet (Ecole Centrale Paris)
Principal examiners :	Dr.-Ing. Maik Maurer (Technical University of Munich)
	Dr. Eric Bonjour (Université de Lorraine)
Examiners :	Dr. Jean-François Boujut (Institut National Polytechnique de Grenoble)
	Prof. Eric Duceau (EADS Innovation Works)
	Dr. Jean-Baptiste Welcomme (EADS Innovation Works)
Guests :	Martine Callot (EADS Innovation Works)
	Yannick Deleris (Airbus)
	Dr. Claude Reyterou (EADS Innovation Works)

A mes parents et ma femme Cecile.

To my parents and my wife Cecile.

ACKNOWLEDGMENT

Je tiens tout d'abord à remercier ma femme Cécile, mes parents et beaux-parents qui m'ont supporté durant cette aventure et notament dans les moments de doute.

Un grand merci à mes collègues d'Innovation Works de Toulouse pour les discussions (scientifiques ou non) et l'ambiance de travail durant ces trois années. Parmis eux, je tiens à remercier particulièrement Claude pour ses remarques piquantes et Jean-Baptiste pour ses avis critiques et ses conseils avisés qui m'ont permis d'avancer dans mes travaux. Cette aventure n'aurait été possible sans la confiance accordée par Arnaud Rivière.

Un remerciement tout spécial à mes deux directeurs scientifiques successifs : Michel Dureigne et Eric Duceau qui ont su poser les bonnes questions (parfois "chiantifiques") tout en me soutenant dans ma démarche de recherche.

Enfin, un grand merci à Mr. Maurer, Mr. Bonjour et Mr. Boujut pour avoir accepté de participer à mon jury de thèse et pour avoir jouer le jeu de cette fin d'année mouvementée.

ABSTRACT

The design of system architectures is a complex task which involves major stakes. During this activity, system designers must create design alternatives and compare them in order to select the most relevant system architecture given a set of criteria.

In order to investigate different alternatives, designers must generally limit their trade studies to a small portion of the design-space which can be composed of a huge amount of solutions. Traditionally, the architecture design process is mainly driven by engineering judgment and designers' experiences and the selected alternatives are often adapted versions of known solutions. The risk is then to select a pertinent but yet under optimal solution.

In order to increase the confidence in the optimality of the selected solution, the coverage of the design-space must be increased. The use of computational design synthesis methods proved to be an efficient way to support designers in the design of engineering artifacts (structures, electrical circuits...). In order to assist system designers during the architecture design process, a computational method for complex systems is defined. This method uses an evolutionary approach (genetic algorithms) to guide the design-space exploration process toward optimal zones. The initial population of the genetic algorithm is created thanks to a computational design synthesis technique which permits to create different physical architectures and allocation mappings for a given functional architecture.

The method permits to obtain the optimal solutions of the stated design problem. These solutions can be then used by designers for more detailed trade studies or for technical negotiations with system suppliers.

RÉSUMÉ

La conception d'architectures de systèmes est une tâche complexe qui implique des enjeux majeurs. Au cours de cette activité, les concepteurs du système doivent créer des alternatives de conception et doivent les comparer entre elles afin de sélectionner l'architecture la plus appropriée suivant un ensemble de critères.

Dans le but d'étudier différentes alternatives, les concepteurs doivent généralement limiter leur étude comparative à une petite partie de l'espace de conception qui peut être composé d'un nombre immense de solutions. Traditionnellement, le processus de conception d'architecture est principalement dirigé par le jugement et l'expérience des concepteurs, et les alternatives sélectionnées sont des versions adaptées de solutions connues. Le risque est donc de sélectionner une solution pertinente mais sous-optimale.

Pour gagner en confiance sur l'optimalité de la solution retenue, la couverture de l'espace de conception doit être augmentée. L'utilisation de méthodes de synthèse calculatoire d'architecture a prouvé qu'elle était un moyen efficace pour supporter les concepteurs dans la conception d'artefacts d'ingénierie (structures, circuits électriques...). Pour assister les concepteurs de systèmes durant le processus de conception d'architecture, une méthode calculatoire pour les systèmes complexes est définie. Cette méthode emploie une approche évolutionnaire (algorithmes génétiques) pour guider le processus d'exploration de l'espace de conception vers les zones optimales. La population initiale de l'algorithme génétique est créée grâce à une technique de synthèse calculatoire d'architecture qui permet de créer différentes architectures physiques et tables d'allocations pour une architecture fonctionnelle donnée.

La méthode permet d'obtenir les solutions optimales du problème de conception posé. Ces solutions peuvent être ensuite utilisées par les concepteurs pour des études comparatives plus détaillées ou pour des négociations avec les fournisseurs de systèmes.

CONTENTS

Part I Introduction 14

1. General introduction 16

2. Research context 17

3. Outline 19

Part II Industrial context 20

1. Systems 22

1.1 Definition 22

1.2 Hierarchy in systems 22

1.3 System architecture 23

2. The system architecture design problem 26

2.1 Description 26

2.2 Stakes 26

2.2.1 An iterative process 26

2.3 Architecture design in the development process 27

2.4 Actors 27

2.5 Characteristics of the problem 28

2.5.1 Complex 28

2.5.2 Multi-objective 29

2.5.3 Collaborative 29

2.5.4 Uncertain 29

3. Thesis industrial objectives 31

Part III Solving the system architecture design problem 33

1. Intuitive methods 36

2. Computational methods 37

2.1 Representation 38

2.1.1 Matrixes and chromosomes 38

2.1.2 Graphs 40

2.1.3 Systems engineering models 40

2.1.4 Grammars 42

2.2 Generation 43

2.2.1 Combination approaches 43

2.2.2 Transformation approaches 44

2.3 Evaluation 49

2.4 Guidance 50

2.4.1 Guidance in evolutionary algorithms 51

2.4.2 Decision-aiding 52

2.5 Classification of some CDS methods 53

3. Thesis scientific objectives 57

Part IV Contributions 58

1. Overview 60

2. Representation 61

2.1 Introduction 61

2.2 Model and meta-model 61

2.3 Meta-model presentation 62

2.3.1 The decision concepts 62

2.3.2 The architecture concepts 64

3. Generation 66
3.1 Architecture synthesis 67
3.1.1 Overview 67
3.1.2 Search algorithm for component chains 67
3.1.3 Introduction of a heuristic for reduction of complexity 68
3.1.4 Implementation of chains 68
3.2 Rules 70
3.2.1 Viability 70
3.2.2 Validity 71
3.2.3 Other rules 71
3.2.4 Example : Architecture synthesis 72
3.3 Evolution 78
3.3.1 Mutation 78
3.3.2 Crossover 81
3.3.3 Example : Crossover 82

4. Evaluation 84

5. Selection 87
5.1 Pareto approach 87
5.2 Preference-based approach 88
5.3 Considering constraints 90

6. Termination and post-analysis 91

Part V Developments: the SAMOA tool 93

1. Introduction 95

2. Architecture 96

3. Modules 97
3.1 Master module 97
3.2 Evolution module 97
3.3 Evaluation modules 98
3.4 Selection module 98
3.5 Solution pool 99
3.6 Human-Machine Interface 99
3.6.1 The Model mode 99
3.6.2 The Process mode 100
3.6.3 The Result mode 100
3.7 Other tools 101
3.7.1 Parsers 101
3.7.2 SysML interface 101

4. Modeling 103

Part VI Application 105

1. Presentation of the cockpit use-case 107
1.1 Introduction 107
1.2 Functions 109
1.3 Realizations 109
1.4 Design process 110

2. Model 112
2.1 Model presentation 112
2.1.1 System 112
2.1.2 Functions 112
2.1.3 Component types 115
2.1.4 Criteria, objectives and constraints 116
2.2 Reference architecture 118

2.3 Results 120

2.3.1 Population initialization 120

2.3.2 Optimization 124

2.3.3 Introduction of crossover 126

Part VII Outcomes and perspectives 130

1. Outcomes 132

1.1 Interests and added value 132

1.1.1 Physical/Functional architectures update 132

1.1.2 Naturally-bounded design-space 132

1.1.3 Compatibility with traditional process 133

1.2 Limitations 133

2. Perspectives 134

2.1 Compatibility rules 134

2.1.1 Capability performances 134

2.1.2 Other types of rules 135

2.1.3 Global initialization 135

2.2 Performances 135

2.2.1 Evolution strategies - Heuristics 135

2.2.2 Mathematical programming 136

2.2.3 Parallelism 137

2.3 Uncertainties 137

2.3.1 Evaluating with uncertainties 138

2.3.2 Selection with uncertainties 138

2.4 Multi-system optimization 138

Part VIII Conclusion 140

Bibliography 149

List of figures 152

List of tables 153

Appendices 153

A. Description of component types 154

B. Availability analysis of the reference architecture 161

C. How to use the SAMOA profile 165

Part I

INTRODUCTION

CHAPTER 1

GENERAL INTRODUCTION

With the development of emerging countries (China, Brazil, India...), the industrial markets see the emergence of new competitors. This new context pushes the traditional manufacturers to put innovation in a central place of their companies. Innovation must permit to differentiate from competitors' products but also to bring more valuable products to the customers.

During the development of new products, innovation can be mainly achieved during early design phases in which the degrees of freedom are numerous. In the field of systems, this corresponds to the architecture design phase during which the building elements of the system and their relations are defined. At this time, the constraints to be satisfied and the needs to cover are known but the main characteristics of the system are not defined. This permits to envisage innovative designs using new technologies, and/or new organizations of existing technologies. Nevertheless, since alternatives can become numerous, methods are needed to support the designers in the exploration of the design-space. Computer-aided design techniques are particularly suited since they are able to handle the complexity of current systems and the number of possible solutions to the design problem.

This dissertation presents the results of the research works that led to the definition of a new design method for complex systems. This method shall allow designers to consider large design-spaces in order to discover innovative and performing design alternatives for their systems.

CHAPTER 2

RESEARCH CONTEXT

The research works started in January 2010 and have been lead within EADS Innovation Works which is the corporate research center of the EADS group (Airbus, Eurocopter, Astrium, Cassidian). The researches have been done in collaboration with the Industrial Engineering lab (Laboratoire Genie Industriel) of Ecole Centrale Paris and with Airbus France.

The research procedure started by the definition of the industrial needs and the analysis of current engineering practices within Airbus R&T teams. This analysis showed the need for support in defining architectures for complex systems. Following the industrial analysis, some existing methods and techniques were assessed to check their pertinence with respect to the actual needs. Some interesting works were assessed but did not cover entirely designers' needs. Consequently, a proposition of method was made to Airbus systems designers to support them during the architecture design problem. The method was applied to a cockpit design problem in order to be able to show the interest of the method to Airbus design processes.

Fig. 2.1: The research context

CHAPTER 3

OUTLINE

The first part of the thesis will introduce the industrial context of the research works. A definition and examples of system architectures will be notably given. The needs perceived through interactions with system designers will also be presented. The expected benefits of the research works will be finally formulated.

An analysis of existing solutions to solve the system architecture design problem will be presented in Part 3. This analysis will show that the domain of research is quite recent but also very active. Nevertheless, the industrial needs stated in Part 2 require the definition of a new method that will be presented in Part 4.

The method uses computational means to synthesize optimal design alternatives. In order to demonstrate the feasibility and the pertinence of the method, a computer program must be developed in order to apply the method and obtain results on a sample problem. The architecture and the main characteristics of this software will be presented in Part 5.

Part 6 will first introduce the use-case which will be used as a test problem for the method. The problem is the design of an aircraft cockpit: a complex and critical system of the aircraft. The method will be applied to this problem using several sets of parameters. This will permit to study the sensitivity of the results to the tuning of the method parameters.

Finally, an analysis of the results will be conducted. Based on this analysis, perspectives for future research works will be presented.

Part II

INDUSTRIAL CONTEXT

CHAPTER 1

SYSTEMS

1.1 Definition

A system is defined as “a combination of interacting elements organized to achieve one or more stated purposes” [INCOSE, 2007]. Systems include hardware, software and humans.

For instance, a car is made of hardware elements (wheels, structure), software elements (control software) and humans (driver [1], passengers).

The purpose of a system is generally described as a set of requirements. These requirements describe the functions to be realized by the system (functional requirements) and that participate to the realization of the system purpose as well as the expected performances of these functions (non-functional requirements).

1.2 Hierarchy in systems

During its development, a system is decomposed in smaller entities called system elements or subsystems. This permits to split the development works into simpler and more manageable design tasks. Each subsystem must participate to the realization of the system functions and must bring value to the final product. The system elements can be themselves considered as systems with their own system elements. The decomposition of systems into subsystems stops when subsystems are simple enough to be considered as a whole or when the subsystems are subcontracted to an industrial partner. An aircraft can be decomposed into three main systems : structure, airframe systems and propulsion (ATA classification Air Transport Association of America [1999]). Then, the "airframe systems" system can be decomposed into its system elements : auto-flight, communications, landing gear... In the same manner, the product can be considered as a subsystem of a higher level system and participates to the creation of value at this higher level. For instance, an aircraft is

1. The driver shall be considered as an element of the car system. Indeed, the main purpose of the car is to move people or objects and this purpose cannot be achieved without a driver.

part of the air transportation system which comprises other aircrafts, airports, Air Traffic Control,. . . .

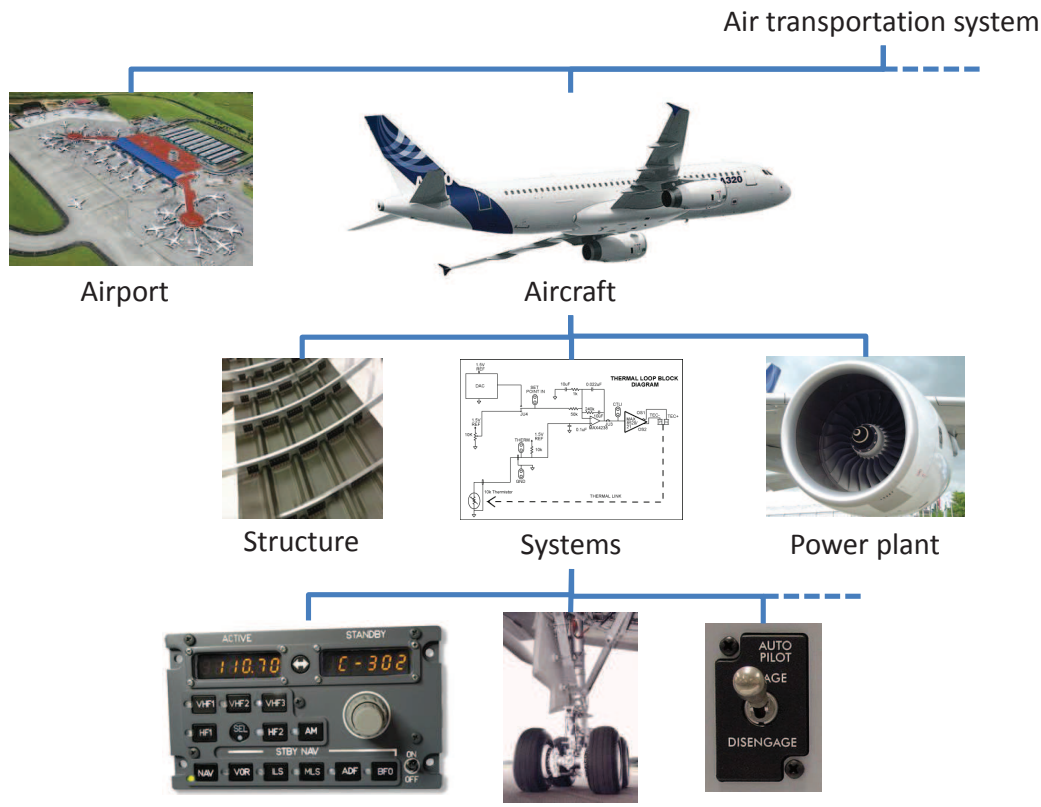


Fig. 1.1: Hierarchy in systems

In this thesis, we consider a system as a single layer object, i.e. we will consider the physical elements of a system as indivisible. Consequently, we will call them components. The assembly of these components is called the system architecture.

1.3 System architecture

A system architecture is the representation of the internal organization of a system. It describes "the components of the system, their relationship to each other and to the environment and the principles governing the design and evolution of the system" [IEEE, 2007].

Ulrich states the following:
"The architecture of the product is the scheme by which the function of the product is allocated to physical components. I define product architecture more precisely as:

1. the arrangement of functional elements
2. the mapping from functional elements to physical components
3. the specification of the interfaces among interacting physical components." [Ulrich, 1995]

This definition introduces the different views describing a system architecture:

- The functional view is a description of the functions to be realized by the system. In this view, functions are linked to each other by hierarchical relations (function to sub-function) or by logical relations (flows, execution sequence. . .).
- The physical view is a description of the components of the system (sub-systems or parts). In this view, components are linked to each other by hierarchical relations (component to sub-component) or by physical connections that will permit components to exchange flows. The functional exchanges can be of different natures (information/signal, matter, energy...).
- The allocation view consists in a mapping between functions and components. The mutual relations of this mapping state that a component realizes (alone) or contributes to the realization (together with other components) of a function.

Figure 1.2 presents different physical architectures for the propulsion system of an aircraft. These different architectures realize the same function : propel the aircraft, but use different technologies and adopt different organizations (serie for a and b, parallel for c). Consequently, their performances will be different, each one having strength on some evaluation criteria (e.g. fuel consumption for a hybrid propulsion system) and weaknesses on others (e.g. maintenance).

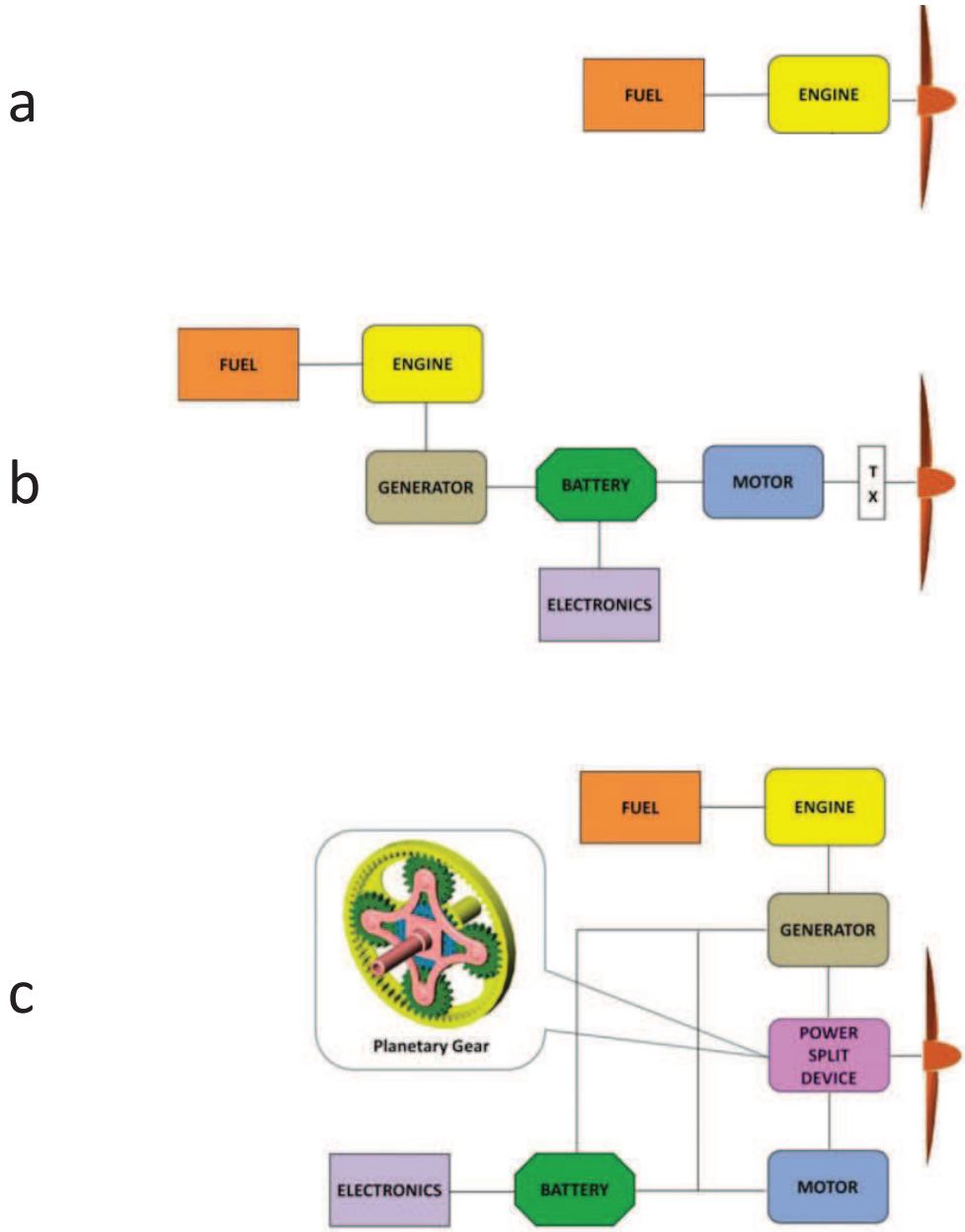


Fig. 1.2: Example of physical architectures for the propulsion system of an aircraft (adapted from Hung and Gonzalez [2012])

CHAPTER 2

THE SYSTEM ARCHITECTURE DESIGN PROBLEM

2.1 Description

The system architecture design process consists in the definition of the main characteristics of the system under consideration. This includes the definition of the system elements, their inter-relationships and their relations to the system environment. Specifications of the system elements are used as technical contracts with suppliers or with their respective design teams.

The definition of the functions of the system, of their relations and of their expected performances leads to the specification of the functional architecture of the system.

Concerning the design of the physical architecture, functions must also be allocated to components i.e. the realization of the system functions must be attributed to one or several components. These components must collaborate and are organized in a way that they are able to realize the function.

2.2 Stakes

The design of a system architecture is an activity involving major stakes since it drives significantly the overall performances and quality attributes of the system. For instance, coming back to the previous example of propulsion system architecture, the difference in performances between a classical and a hybrid propulsion system can be quite important. When one of these architectures is adopted, the attainable performances are significantly bounded, whatever the actual implementation (detailed specification of system elements) of this architecture is.

2.2.1 An iterative process

The process of system architecture is often viewed as a linear transformation of a functional architecture into a physical architecture. Nevertheless, some decisions made on

the physical architecture have impacts on the functional one i.e. the use of some technologies induce some functions that must be realized by the system.

For instance, some components may produce heat and may require a cooling function. This function is part of the functional architecture due to the presence of an instance of the heating component in the physical architecture. In order to have a complete and coherent architecture, this function must be allocated to a set of components capable of realizing the induced function.

Also, decisions made on other systems may have impacts on the architecture of the system under investigation. Thus, iterations are necessary at the product level in order to guarantee the coherence of the different systems.

2.3 Architecture design in the development process

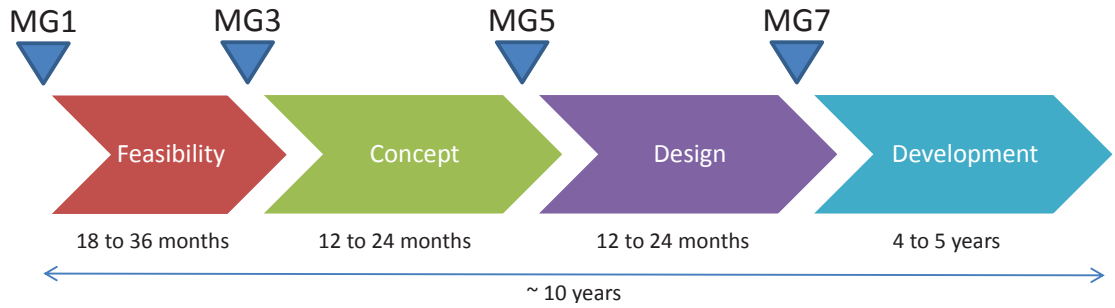
During the development of a program, degrees of freedom are progressively fixed following the maturity gates (MG) or milestones go-through process. At each maturity gate, decisions on aircraft design are taken and will constraint the following design activities. The system architecture design happens in an early phase of the development process when the main characteristics of the system must be fixed. In an Airbus context for instance, it approximately goes from MG1 (feasibility study) to MG5 (conceptual design). In an the space industry, it goes from phase 0 where constraints are identified and systems concepts (concepts of operations, high level architectures...) are created until phase C where the final architecture is frozen.

2.4 Actors

Architecture design activities are conducted by a team of designers with complementary expertise. The design team is in charge of:

1. defining and refining requirements,
2. identifying interfaces and impacts to/from other systems,
3. defining design alternatives,
4. selecting the baseline architecture on which the detailed development activities will be based.

If the system is subcontracted as a whole, the architecture design is performed by the system supplier. Nevertheless, the design team must be able to criticize and challenge the design proposed by the subcontractor. For this, different alternatives must be created and analyzed in order to better understand the design space and the performances reachable by the system.



(a) Airbus development process

(b) Space industry development process (Source: [NASA, 2007; Guerra, 2008])

Fig. 2.1: Development processes in the aerospace industry

The architecture design problem presents several characteristics that make it one of the most critic and complicated phase of the development process.

2.5 Characteristics of the problem

2.5.1 Complex

The architecture design problem is a complex task as designers generally have a huge amount of possibilities (design alternatives) to realize the system functions. The process of selecting the best alternative (baseline) is even more complex.

Complexity of the problem also lies in the complexity of the manipulated objects. Indeed, systems are often complex because of:

- their structure (number of components, connections)
- their behavior (emergent properties)

Designers must understand the interactions between the system elements and with other systems in order to guarantee the required performances and to avoid any undesirable emergent behavior.

2.5.2 Multi-objective

The architecture design process includes high-impact decisions on the system structure. These decisions importantly influence the performances of the system and the customer perceived quality of the product. These decisions are not straightforward as they generally consider several objectives. Moreover, some of these objectives may be conflicting i.e. the improvement of an objective causes the degradation of another one. An example of conflicting objectives is reliability and costs as reliable equipments are generally more expensive. In these case, designers must make trade-offs and select the best compromise in terms of performances.

2.5.3 Collaborative

The architecture design process at a system level (e.g. propulsion system) does not aim at specifying the best system. Its goal is to find the system architecture that will lead to the development of the best product (e.g. aircraft) within the project budget (time and cost). Indeed, the optimal product may not be an assembly of the optimal systems considered separately. In order to integrate their study in a product context, the design team must take into account the impacts of its architecture definition on other systems and the impacts of other systems on their system [Kroo, 2004].

There also exist collaborations within design teams since several designers may work together on the system definition. Each designer generally have its own experiences, knowledge and domains of interest. This is often a great deal to merge these different viewpoints to obtain a consensus for the adoption of the final architecture.

2.5.4 Uncertain

In the system development process, the architecture design phase occurs during the first months of the project and are generally initiated during the research projects preceding the actual product development. At this time, several types of uncertainties are present :

Technological uncertainty The maturity of technologies and their performances at the time of detailed specification are uncertain and must be anticipated. In order to estimate performances of existing technologies through continuous improvement, designers generally turn to an extrapolation of performance improvement through time (e.g. Moore laws for computer power). Nevertheless, this estimation is rough as evolution of technologies are not always continuous and may be subject to sudden improvements.

Market evolution During the development project, the product market may evolve. This fact can have impacts on the definition of value by the designers as objectives' relative importance may change.

These uncertainties can largely influence the decision (e.g. risky situation) or the quality of the decision perceived at the end of the project (e.g. under-optimal adopted solution). Robust optimization techniques [Mulvey et al., 1995] can be used to obtain solutions that are robustly optimal with respect to variations of uncertain parameters.

The architecture design problem is a complicated problem. To solve it efficiently, current processes reach their limit. The need for a method enabling the design-space exploration can be highlighted and industrial objectives of this method can be stated.

CHAPTER 3

THESIS INDUSTRIAL OBJECTIVES

The design of systems architectures is a key task of the development process as it largely influences the quality of the final product. This stage requires the generation of design alternatives by the system design team. Currently, this task is mainly performed manually through brainstorming and/or analysis of existing product architectures. Given the complexity of the current aeronautical systems, the design-space can be huge and only a narrow part of it can be reasonably explored this way. In order to explore the design space in a wider way, a method to generate design alternatives for the design problem is needed. This will be the main objective of these research works. From an industrial viewpoint, the proposed method shall:

Objective 1 generate a number of pseudo-optimal design alternatives for the physical architecture of complex systems.

- Objective 1.1 synthesize architectures for systems with multiple functions, possibly exchanging flows between them.
- Objective 1.2 be able to allocate several functions to a single system element.
- Objective 1.3 be able to fulfill safety constraints by creating redundancies.
- Objective 1.4 be able to add induced functions to the functional architecture and to allocate them to system elements.

Objective 2 require a limited and simple design knowledge elicitation process. This objective is quite subjective but the idea behind it is that the user of the method must not be constrained to describe explicitly the design space.

Objective 3 consider existing and emerging technologies. Particularly, the design knowledge for new technologies must be able to be acquired by the method.

Given a capability to generate design alternatives, a number of architectures can be obtained. It is then necessary to be able to select the best ones to allow a final decision to be taken or to permit further studies to be carried out manually on a small set of alternatives. This selection process is a multi-objective decision problem and requires being able to:

- formalize objectives and constraints
- assess architectures
- define the worth of an alternative
- treat the performance data to select best architectures

Consequently, the proposed method shall additionally be able to :

Objective 4 assess the performances of architectures with respect to diverse criteria (e.g. mass, costs, safety...).

Objective 5 compare different alternatives based on their performances and on the designers' definition of the worth of an alternative.

Objective 6 select the best architectures to be presented to the designers' for final decision-making.

In the next part, the means currently available to solve this design problem will be presented.

Part III

SOLVING THE SYSTEM ARCHITECTURE DESIGN PROBLEM

III. SOLVING THE SYSTEM ARCHITECTURE DESIGN PROBLEM

Researchers in design engineering have interested since a long time to the problem of architecture design. Their researches led to the definition of several methods ranging from intuitive, based on an organization of knowledge, to computational methods, based on computational algorithms.

CHAPTER 1

INTUITIVE METHODS

The first methods that have been formalized to solve the system architecture design problem are intuitive methods such as brainstorming [Osborn, 1953], brainwriting [Dennett, 1975], TRIZ [Altshuller, 1999] or the 635 method ([Rohrbach, 1969; Shroer et al., 2010]).

These methods rely on the creativity of a group of experts, on their experiences and on their own knowledge to produce solutions to a given problem. These methods are interesting as they permit to mix these different experiences to produce innovative solutions.

The main limitation of these methods is their capability to handle complexity. Indeed, humans are limited in their understanding of complex problems. As they are only based on human cognition and interactions, complex design problems cannot be addressed using intuitive methods.

Intuitive methods permit to generate innovative solutions for problems. Nevertheless, they are limited in the complexity of the problems they can address and they lack of a formal frame that gives confidence in the coverage of the solution-space.

To remedy to these weaknesses, methods making profits of the power of computers were designed to support the design of systems architectures.

CHAPTER 2

COMPUTATIONAL METHODS

The birth of computational methods was based on the assessment that only computers can handle complexity and bring formal solutions to a design problem. Computational design synthesis (CDS) methods permit to generate solutions to a design problem based on a formal description of it, by use of computational techniques.

The CDS methods are numerous. In 2005, Cagan et al. presented a framework to classify these methods [Cagan et al., 2005]. This framework is based on 4 steps: representation, generation, evaluation, and guidance (Figure 2.1).

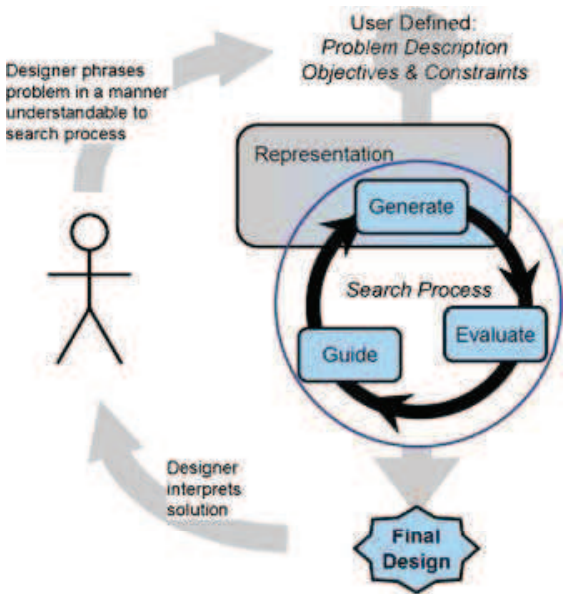


Fig. 2.1: Illustration of Cagan et al. framework

2.1 Representation

The first step of a CDS method is to describe the problem on which the method must be applied. This constitutes the input of the method. The choice of a representation is very important as it will condition the possibility to use certain means for the 3 following steps (Generation, Evaluation, Guidance). Also, it can induce restrictions on the possibility to represent every aspects of the design problem.

The representation of a real artifact (in our case, an architecture) is called a model. In [Fritzon, 2004], a model is defined as "anything an experiment can be applied to in order to answer questions about the system". This definition is, is a sense, quite restrictive as it does not include sketches, drawings and even 3D models, that do not necessarily permit to realize experiments but are means of representation of the modeled object. A more complete definition of a model is "a representation of one or more concepts that may be realized in the physical world" [Friedenthal and Moore, 2011].

The following lines will introduce the major types of models used in CDS methods.

2.1.1 Matrixes and chromosomes

In the late 40s, Zwicky proposed a method to enhance designers creativity [Zwicky, 1948]. In this method, a solution to a given problem is represented as a set of characteristics. Each characteristic has a set of options. The method generates solutions by assembling options for each characteristic. Zwicky proposed to represent the problem under the form of a table in which rows are characteristics of the solutions and options are cells of the corresponding row. This representation is known as morphological box or matrix. Figure 2.2 is a morphological matrix for a motorcycle. The dotted lines represent two potential solutions to the design problem.

Function	Solution principles		
Propulsion	Combustion engine	Electrical motor	Hybrid propulsion
Store electrical energy	Lead battery	NiCd battery	Li-ion battery
Store gasoline	Gasoline tank	No tank	
Support driver	Steel frame	Aluminum frame	Carbon fiber frame
Brake	Disk brake	Drum brake	Regenerative electrical brake

Fig. 2.2: A morphological matrix for the motorcycle design problem (from Olvander et al. [2009])

More recently, some methods inspired from morphological analysis represented the problem and its solutions under the form of matrices. The problem is formalized by modeling the functional architecture of the system as a function-to-function matrix which describes the system functions and the flow exchanges between these functions. The compatibility of components can also be described as an input of the method under the form of a component-to-component matrix. The solution to this problem is represented as a function-to-component matrix (Domain Mapping Matrix) that represents the allocation of a function to a component. Additionally, a component-to-component matrix (component Design Structure Matrix) can represent the physical architecture showing the connections between components.

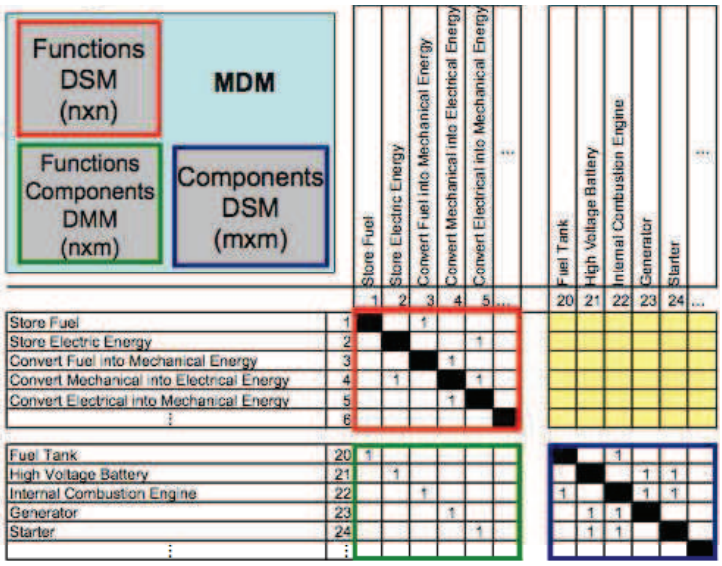


Fig. 2.3: Example of component DSM, function DSM and DMM (from [Gorbea et al., 2008])

The chromosomes, used in genetic algorithms (see Section 2.2.2), are very similar to morphological matrices as they represent characteristics of the solution with their options.

The main strength of a matrix representation is that is is simple and easily understandable. Also, this representation permits to employ mathematical techniques (matrices operations and manipulations, constraint programming).

But matrices are quite limited in their modeling power as they can only represent (qualitative or quantitative) relations between pairs of objects. Also, the size of matrices and chromosomes is fixed. This forces the designer to fix a priori the size of the solution i.e. the maximum number of each type of components that can be in the architecture, and can lead to ignore potential interesting solutions. At the contrary, setting to a two high

size of matrices or chromosomes leads to a lower performance of the resolution technique since many variables may be superfluous.

2.1.2 Graphs

A representation that is often used to represent system architectures is the graph representation. A graph is a set of nodes linked by arcs (directed or not, valued or not). Graphs can represent connections between functions, between components and between components and functions (allocations). This representation is very close from the matrix one as graphs are often represented as matrices (incidence, adjacency, Laplacian...).

This type of representation is particularly used because of its visual representation. In some specific cases, graph theory can be used to solve design problems modeled as graphs. Graphs can have variable size by addition or removal of nodes and connections. This property in used in graph-grammar approaches where rules transform an initial graph representing the architecture (see Sections 2.1.4 and 2.2.2).

2.1.3 Systems engineering models

Dealing with system architectures, systems engineering models are widely used. Systems engineering models are made of objects and of relations between them. These objects and relations can be of different types. An object of a model can be a function, a component, a port or any concept that must be considered in the study. These concepts are defined in a meta-model and implemented by a modeling language.

The System Modeling Language SysML [OMG, 2010] is used in several CDS methods. SysML permits to represent the different aspects of the system (structural and behavioral) using visual constructs. Some recent research works [Masin et al., 2011] attempt to extend the language capabilities to model variability in systems in order to describe design alternatives (e.g. different choices of technologies for a component, different number of components).

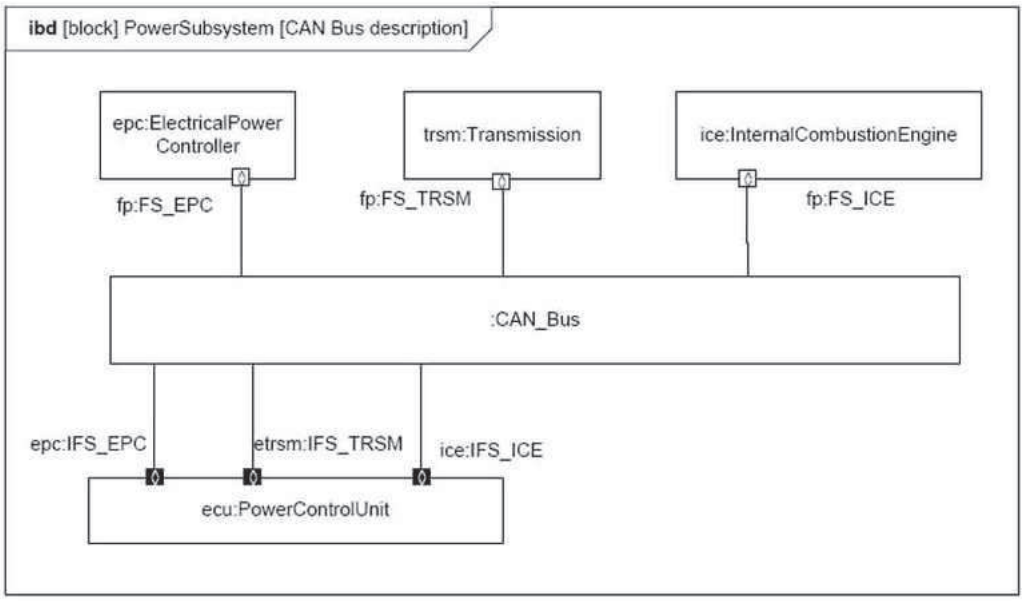


Fig. 2.4: Example of diagram in SysML

AADL [Feiler et al., 2006] is also used to represent architectures of large-scale software-intensive embedded systems.

Other formalism such as Modelica [Modelica Association, 2012] in [Rai, 2011] or bond-graphs in [Seo et al., 2003] are used is CDS methods for the design of multi-physical systems. These formalisms permit to represent energy flows between the components of a system and energy transformation performed within the component thanks to differential algebraic equations.

Systems engineering models can be viewed as graphs with different types of nodes (functions, components...) and of edges (allocations, connections...). Thus, this representation is very close to a graph representation and techniques applicable to graphs are generally also applicable to models.

The representation of architectures under the form of models presents several advantages among which :

- Models are able to capture large amount of complex information.
- Models facilitates communication through a common language and visual means.
- Complex analyses can be applied to models.

Given these advantages of using models in engineering activities, the Model-Based Systems Engineering (MBSE) initiative [Estefan, 2007] encourages the intensive use of models and the replacement of document-based processes by model-based ones. Nowadays, a large number of companies are introducing this philosophy in their processes.

The three previously presented types represent directly the modeled artifact : the "what". At the contrary, grammars permit to represent the way the artifact was created : the "how". Knowing the starting point of the creation process, they permit to indirectly represent the artifact itself.

2.1.4 Grammars

The main idea of grammars is that an artifact can be represented under the form of an initial state and of the transformations that lead to the represented artifact.

The representation, inspired from the works of Chomsky in linguistics, was first adopted by Stiny in the 70s [Stiny, 1972] for the description of shapes. The shape-grammar describes shapes as a set of transformations performed sequentially on an initial shape. Transformations are described as production rules that permit to transform a given shape (left hand side) into another shape (right hand side).

The graph-grammar representation is a generalization of linguistic and shape grammars working with graphs. A graph is represented by an initial graph and the production rules applied to obtain it. The production rules are described as two graphs : a left-hand side (initiator) that will allow the application of the rule and a right-hand side (generator) that will replace the left-hand side in the new graph. Graphs are composed of nodes, belonging to a so-called vocabulary that defines the nature of nodes, and of edges between these nodes.

In [Alber and Rudolph, 2004], a graph-grammar was defined for the design of truss-like structures. In this grammar, the vocabulary is a set of 6 structural elements (A, 1,2,3,4,T). The element A constitutes the initial graph and the element T is a termination node i.e. it is only present in the right-hand side of rules and thus terminates the generation process. A list of production rules is also declared (see Figure 2.5).

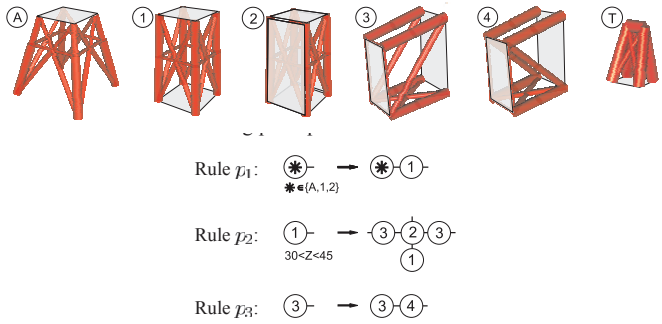


Fig. 2.5: Example of a graph-grammar for the design of truss-like structures (from [Alber and Rudolph, 2004])

Grammars can also be used with engineering models such as SysML [Kerzhner and Paredis, 2009] or Modelica [Rai, 2011]. The grammar is then similar to a graph-grammar but can also work on model data (attributes, relations...).

The main interest of grammars is that they permit to trace the generation of an artifact and thus to go back easily to a previous state. This notation is associated to rule-based CDS methods (see p. 44).

2.2 Generation

The generation step of a CDS method consists in generating design alternatives that are solutions to the design problem given as input of the method. For this step, one can distinguish combination and transformation approaches.

2.2.1 Combination approaches

When the solutions to the design problem can be described as a set of characteristics with options for each, a simple way to get a particular solution is to select an option in each characteristic domain. For instance, in Figure 2.2 (38), the combination {*Hybrid propulsion, Li-ion battery, Gasoline tank, Carbon fiber frame, Regenerative electrical brake*} defines a solution. Generating all solutions of the problem can be done by generating all combinations (thanks to a sequence of "for" loop in a computer program for instance). Nevertheless, some options of a characteristic may not be coherent with options in other characteristics (mutual exclusion) or, at the contrary, some options may imply the adoption of some options for other characteristics. Also, some solutions may not be desirable for other reasons (structural properties, low performances...). To handle these type of constraints, constraint programming is widely used.

Constraint programming (CP) is based on the construction and exploration of a tree of solutions. For each characteristic, each option is represented as a node linked to the upper node (an option of the previous characteristic) by an arc. The efficiency of CP algorithms lies in mechanisms that permit to avoid the exploration of the whole solution tree. For instance, the depth-first search algorithm gradually assigns values to variables (solution characteristics) and reduces the domain (possible options) of the next variables to the options that fulfill all constraints. Thus, when a conflict is detected (i.e. when the domain of the next variable is empty), the algorithm aborts the exploration of the branch and restarts from an upper state (backtracking). Other techniques, such as constraint propagation, permit to limit the number of tree branches considered during the solution search and to consequently save computation time.

In the literature, one can notice two strategies for the definition of design constraints :

- The first strategy consists in asking the designer to formalize its design knowledge by defining design constraints explicitly. This process can be quite heavy as design constraints must describe every possible cases of elements assembly of a design problem. Only considering component-to-component connection constraints, N components will require $(N-1)^2$ constraints. The omission of a constraint can cut a whole portion of the design-space, possibly enclosing the optimal solution, or, at the contrary, can consider solutions that are not acceptable.
- The second strategy consists in deducing constraints from existing architectures [Bryant et al., 2005; Kurtoglu and Campbell, 2009]. For this, a design repository gathering different technical solutions must be constituted and analyzed to get possible association of elements (functions and components). Compatibilities between the different elements of the problem are then injected in the synthesis process. Proceeding so, historical occurrence of elements association can be taken into account for guidance and/or decision-aiding [Bryant et al., 2005]. The main limitation of this approach is that, since the analysis is performed on existing architectures, the design knowledge for new technologies cannot be acquired.

2.2.2 Transformation approaches

Rule-based approaches

Rule-based approaches are based on an iterative modification of an initial object to generate solutions to a design problem. These techniques are generally associated to grammar representations (Section 2.1.4).

Rules are a formal representation of the knowledge of designers. These rules describe how designers reason to create architectures or to modify existing architectures.

Rule-based CDS methods have been applied to a wide range of applications as they provide flexibility to address different types of design problems.

Shape-grammar permits to generate paintings and sculptures. The vocabulary, composed of basic 2D or 3D shapes, is used in production rules that permit to generate more complex objects.

In the industrial field, such methods have been applied to different design problems : pylons for power lines [Alber and Rudolph, 2004] , micromechanical resonators [Bolognini et al., 2007], vacuum cleaners [Wyatt et al., 2009],... Many of these applications only address the physical aspects of the system i.e. they only consider an assembly of components without any link to the functional aspects.

Kurtoglu and Campbell [2009] proposed a method based on a grammar and on functions-to-components rules that permit to generate system architectures Eligible parts of the functional architecture (LHS) are replaced by structures of components (RHS). The algorithm applies rules until all functions are replaced by components (in other words until all function are allocated). If several rules are eligible (e.g. functions can be realized by different graphs of components), the algorithm can create design alternatives for the system architecture. The function-to-components rules permit to establish a link between the functional and physical architectures (allocations).

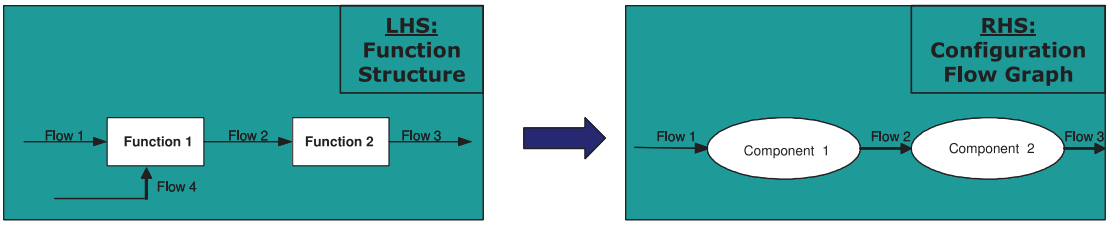


Fig. 2.6: An hypothetical function to component rule (from [Kurtoglu and Campbell, 2009])

The *booggie* software [Technische Universitat Munchen, 2012] developed by the Technical University of Munich implements object-oriented graph-grammar. The tool permits to declare a vocabulary, an initial graph, production rules and to apply a sequence of rules to produce a new artifact.

The main interest is that with a limited set of simple rules, the process can generate a large number of different designs ranging from simple to very complex ones. On the other hand, forgetting a rule may lead to ignore a whole part of the design-space possibly containing interesting solutions. Also, this technique is very similar to local optimization techniques as new solutions are considered only in the neighborhood (in the transformation sense) of the current solution. Thus, obtaining a solution may require a high number of transformations and consequently an important computation time. In order to solve this

problem, techniques issued from the Artificial Intelligence field, such as Genetic Algorithms or Multi-Agent Systems, constitute a smart way to iteratively generate designs and to progressively improve their quality (optimization).

Genetic algorithms

Genetic algorithms are based on natural evolutionary principles. A solution (or individual) is described using a chromosome. This chromosome is constituted of genes that represent characteristics of the solutions. Genes can take different forms: integers, floats, strings, objects... A first population is created by randomly choosing values for each gene. Each individual is evaluated and a selection of the best individuals is operated based on a fitness function. A new population is then generated by applying genetic operators (mutation or crossover) to selected individuals. This process is iterated until a termination criterion (number of population or convergence indicator) is reached.

Genetics algorithm are mainly used to find the best combination of components in a given system architecture [Gubitosa et al., 2009]. The genes of the chromosome represent the abstract elements of the system (e.g. a piston) or parameters of these elements (e.g. the flow of a pump). Each gene has a domain : elements options (e.g. "32/22 piston") or parameters ranges (e.g. $flow \in [3; 10]$). A solution is a chromosome where each gene has a particular value comprised in their domain.

In [Sallak et al., 2002], genetic algorithms are applied to a graph of components. The genes of the chromosomes correspond to the connections between available components. The optimization process permits to find solutions which respond to safety requirements at a minimal cost.

Initiated by Holland [Holland, 1975], genetic algorithms are now widely used in different domains and notably in design [Renner, 2003]. They are quite simple to understand and to use and proved to be very robust in solving different types of problems: discrete/-continuous/mixed, convex/non-convex, discontinuous....

Genetic algorithms also present the advantage of being able to be parallelized Nowostawski [1999]. Using parallelized genetic algorithms, a set of computers are used to perform the optimization in order to reduce the computing time or to increase the number of considered individuals (number of populations, population size).

The main limitation of genetic algorithms is that they work on a fixed-size problem i.e. a chromosome. This requires to fix a priori the number of considered solutions and, concerning the architecture design problem, the maximum number of components in the architecture.

Genetic programming

Unlike genetic algorithms, genetic programming work with trees and consequently do not have a fixed problem size. Genetic programming algorithm exchange branches of different tree to mimic genetic algorithm crossover and change the composition and structure of some branches to perform mutations.

Genetic programming is mainly used for the design of computer programs. A set of elementary programs and variables are assembled to form a complete program realizing a given function.

For technical systems, genetic programming is mainly used in association with rule-based approaches. The application of rules forms a tree on which the genetic programming principles are applied. This permits to progressively converge toward the sequence of production rules that optimize objectives under constraints. For instance, in [Seo et al., 2003], genetic programming is used to modify the tree of functions (add, insert, replace, arithmetic) applied to an initial bond-graph embryo to obtain a given solution.

In [Koza et al., 1996], genetic programming is used in the same manner to synthesize electrical circuits. The fact that the optimal synthesized circuits infringe on existing patents shows the power of genetic programming to synthesize performing and innovative designs.

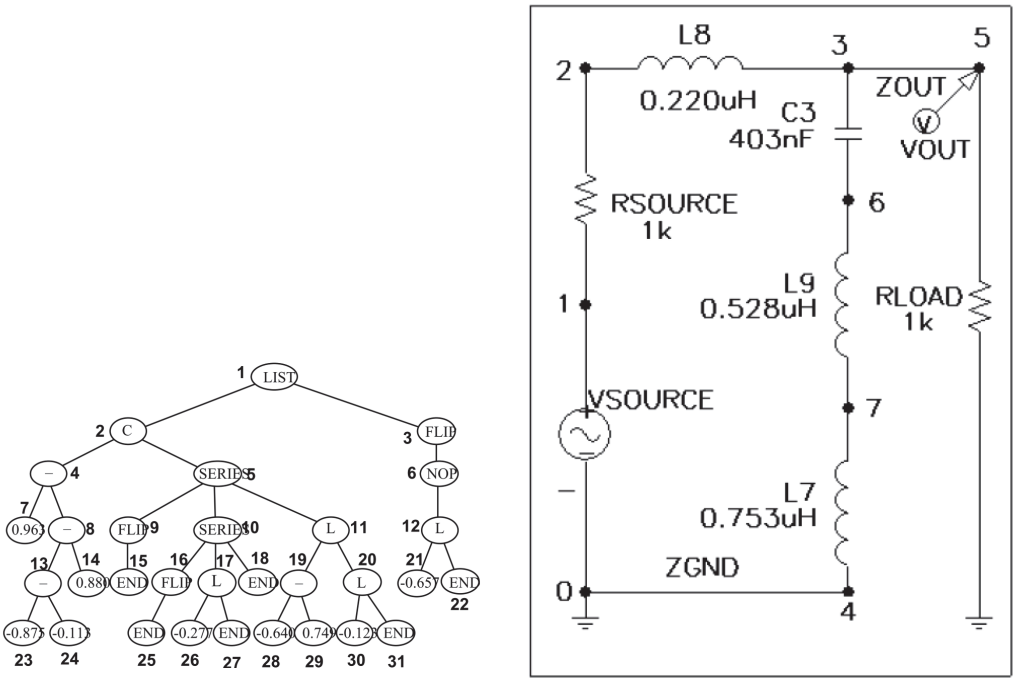
Multi-Agent Systems

Issued from artificial intelligence domain, Multi-Agent Systems (MAS) are systems composed of agents collaborating to reach goals. Agents are autonomous software elements that are able to communicate/negotiate with other agents. Thanks to these interactions, agents are able to find a solution that:

- satisfies their local constraints
- contribute to the coherence of the global solution (global constraints).

MASs were used for optimization of system architectures ([Welcomme, 2008]). In these approaches, agents represent elements of the system (components, disciplines...), have degrees of freedom and constraints. When an agent has a violated constraint, it communicates to its surrounding agents in order that they find a solution to make this constraint satisfied. Thanks to this collaborative behavior, the MAS is able to find a solution. Nevertheless, these methods only consider the behavior of agents as degrees of freedom and not their organization (the architecture).

A MAS method tackling the problem of architecture design synthesis is the A-Design System ([Campbell et al., 2000]). In this approach, agents have the possibility to modify



(a) Program tree *(b)* Circuit

Fig. 2.7: Example of an electrical circuit and its program tree (from [Koza et al., 1996])

the architecture (add/remove components, configure components...) following their own rules in order to find optimal solutions.

MAS optimization methods are quite efficient since they can exploit heuristics to optimize given criteria. On the other hand, they are problem-specific and require the elicitation of design knowledge.

The above presented optimization permit to explore the design-space in a smart way, looking for optimal solutions. In order to know the worth of a solution, it is necessary to go through an evolution process.

2.3 Evaluation

The evaluation step of a CDS method consists in assessing the worth of every generated alternatives. This is done by assessing the performances of each alternative along one or several evaluation criteria.

Of course, since CDS method aim at generating a large amount of design alternatives, this task must be automated. Depending on the problem, its representation and the considered evaluation criteria, different techniques can be employed. We give here after some examples of evaluation techniques.

First, many criteria can be evaluated using simple (empirical) laws which are linear relations between the characteristics of the architecture and its overall performance. Criteria such as mass and cost are examples of criteria that are traditionally computed by these types of evaluations. The use of heuristics is very convenient as they are easy to set up and require low computation type. On the other hand, they only provide estimations of the real system performances as they represent simplifications of the real system.

When more detailed analyses are needed, several computational techniques are generally available depending on the criterion to assess.

To assess signal-based system properties such as end-to-end transmission times in an information system, discrete-time simulation can be used. The simulation evolve the state of the system depending on occurring events (internal or external). Formal methods [Scharbarg and Fraboul, 2010] (model-checking, network algebra, trajectory...) can also be applied but often require large computation times which makes them unusable for optimization purpose.

For physical systems, 0D-1D energy-based simulations can be used. For this, languages such as Modelica or VHDL (and their associated platforms) or simulation tools such as Amesim, Flowmaster or SABER can be used. During simulation, components of the system exchange energy flows and signals. These flows and signals are quantified using the

components' behavior described under the form of equations (acausal form) or assignments (causal form).

Geometrical and matter-related criteria are generally evaluated through the use of Finite Element Methods or Finite Volume Methods. These methods can evaluate mechanical strength of structures [Alber and Rudolph, 2004], properties of MEMS[1] [Bolognini et al., 2007], propagation of fluids, thermal properties...

RAMS (Reliability, Availability, Maintainability, Safety) criteria are generally evaluated thanks to reliability graphs [Robin et al., 1996] and their associated computation methods. A reliability graph represents the elements of the system and their relations. Algorithms permit to find minimal paths, minimal cuts and, finally, the reliability of the system. This technique assumes that components have only one failure mode and that failures do not propagate in the system. When a more detailed analysis is needed, safety models can be used. The Altarica language [Arnold and Point, 1999] permits to construct such models by describing the system structure, components failure modes and failure propagations. Then advanced algorithms permit to compute the performances of architectures on the considered RAMS criteria.

Since some evaluations may require large computational time (e.g. Finite Element Method, Computational Fluid Dynamic or model-checking) and since optimization techniques often requires to evaluate a large amount of alternatives, the choice of the evaluation technique must be done carefully in order to have reasonable computation times. Also, different techniques can be used sequentially during the process : simple analyses to initiate the search and more complex ones when interesting zones of the design-space are identified.

The computed performances are used to guide the search of new solutions.

2.4 Guidance

The guidance step permits to influence the generation step in order to generate improved designs. It acts as a feedback loop on it. Based on the information (the performances of generated alternatives) gathered by the evaluation process (sensor), the guidance step will control the generation process in order to obtain better designs (output). This step is necessary when the design-space is too large to permit the generation and evaluation of every solutions, which is the case in most system design-problems.

1. Micro-Electro-Mechanical Systems

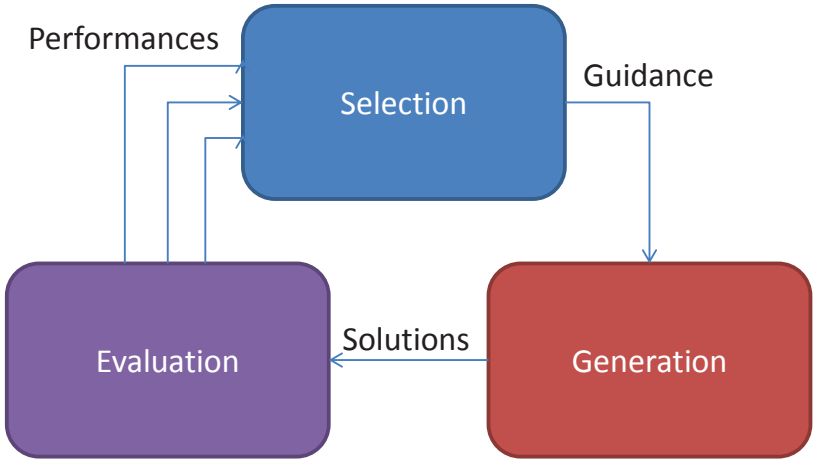


Fig. 2.8: The guidance process

2.4.1 Guidance in evolutionary algorithms

When evolutionary principles are used to generate design alternatives, guidance is performed through the analysis of previous results (i.e. performances of generated solutions) and the application of evolution mechanisms to the best solutions of the previous iteration. For this, the worth of an alternative must be defined (absolutely or relatively to other alternatives) to allow the selection process to select the best alternatives of the previous population. This definition can be stated in different terms as described here after.

Fitness function

In genetic algorithms, a fitness function representing the worth of alternatives permits to favor the reproduction of good individuals. The fitness function is a mathematical function (generally a weighted sum) of the considered criteria. Determining the fitness function (e.g. determining the relative importance of criteria) is not straightforward and may lead to different results. Even if some methods support the computation of these weights (Analytical Hierarchy Process [Saaty, 1980], Simple Multi-Attribute Rating Technique [Edwards, 1977]...), other selection processes permit to avoid defining a-priori fitness functions.

Pareto approaches

A classical manner to deal with multi-objectives problems is to use the Pareto dominance rule. Following this rule, a solution A is said to dominate a solution B if it is no

worse than B in all objectives and is strictly better than B in at least one objective. Several algorithms among which NSGA-II [Deb et al., 2000] and SPEA2 [Zitzler et al., 2001] use this rule to select efficient solutions and estimate the Pareto front of the problem. An interest of this approach is that it permits to see the relationships between the different objectives (i.e. how much a performance degrade when another one improves) thus allowing the decision-maker to better understand the problem .

On the other hand, the solutions obtained are generally spread all along the Pareto front which contains solutions that do not fit the decision-maker preferences (generally extreme solutions).

Preference-based approaches

Some innovative multi-objective optimization approaches, arose for Operational Research community, propose to aggregate objectives under a mathematical function that takes into account decision-makers preferences. An example of such an approach is used the NEMO algorithm [Branke et al., 2009]. In this algorithm, the performances $G_i \in G$ on every objectives are aggregated under a utility concept:

$$U(x) = \sum_{i=1}^{n} u_i(x) \tag{2.1}$$

where u_i are non-increasing marginal value functions, $u_i : G_i \to \mathbb{R}$.

Preferences of the decision-maker are elicited all along the process by comparing pairs of alternatives. Then, the compatible utility functions are computed thanks to linear programming and robust preference-based dominance relations are built. Using this approach, designers directly influence the design-space exploration performed by the generation algorithm. The solutions obtained at the end of the optimization algorithm are close to the decision-maker expectations. Nevertheless, the implementation of this approach requires that the designers are available all along the optimization process. Also, it requires more computation time than Pareto approaches because of the time to solve a number of linear problems.

2.4.2 Decision-aiding

If the size of the design-space allows the generation of all solutions or if the optimization process leads to several alternatives, designers shall be supported to choose the solution to adopt. Indeed, in the case of multiple criteria, such decision may not be straightforward as no unique optimal solution (i.e. a solution that is optimal on each criterion) may exist.

In order to support decision-making in case of multi-attribute problems, a wide variety of methods were created during the last century. These methods are called multi-criteria decision analysis (MCDA) methods (see [Guitouni and Martel, 1998] for a review).

An MCDA process is basically made of 4 steps:

1. Collect alternatives
2. Collect criteria to take into account
3. Establish the performance table
4. Aggregate performances

During the last step, preferences of the decision-makers can be elicited and used for the choice of a solution or for the ranking/sorting of several solutions.

Complete aggregation methods

These methods aggregate all criteria under a single indicator. The decision is then based on the comparison of this indicator for all alternatives. In these methods, compensation is allowed i.e. a bad evaluation on one criteria can be compensated by a good one on another criteria. This is not always representative of the decision logics.

Examples of such methods are Analytic Hierarchy Process [Saaty, 1980] or TOPSIS [Yoon and Hwang, 1995].

Outranking methods

These methods are based on a pair wise comparison of alternatives along every criterion taken individually. Then, preference relations are built by considering the mono-criterion outranking relations. This type of method presents the advantage to consider individually every criterion limiting the compensation effect but the results and their exploitation can be quite complex.

Examples of such methods are Electre [Figueira et al., 2005] or Rubis [Bidorff et al., 2008].

2.5 Classification of some CDS methods

In this section, we demonstrate the use of the CDS framework by classifying some CDS methods with respect to their characteristics on the 4 steps (Figures 2.1 and 2.2).

This analysis permits to highlight that few references propose a complete CDS process (5 of 13). Other methods only generate designs randomly, and do not search best solutions,

or generate every possible designs (full-factorial), which can lead to huge computational costs depending on the size of the design-space (i.e. the number of solutions). Also, these methods make the work of designers even heavier than working with classical design methods as they provide them with a large amount of raw information (the solutions) an do not help them in choosing the final solution(s).

Another observation is that seldom are the methods considering functional aspects of architectures in the process: [Kurtoglu and Campbell, 2009; Holley, 2011]. The first one is based on graph-grammar, and thus, requires to describe explicitly, for each function, every means to realize it. This can become a huge work and can be prone to errors. The second one only considers independent functions, uses very simple evaluation means (Quality Function Deployment) and does not use optimization techniques but a full-factorial approach.

Given the industrial needs presented in the first part and given the state-of-the-art on the topic, it appears that it is necessary to develop a new method to support designers during the architecture definition phase. In the next part, the scientific objectives of this method will be highlighted.

Reference	Representation	Generation	Evaluation	Guidance	Comments
Wyatt et al. [2012]	Graph	Depth-First-Search	Any	-	Starting points of the search are known architectures
Strawbridge et al. [2002]	Matrices	Matrices computations	-	-	The method only suggests components for a product based on their historical occurrence.
Bryant et al. [2005]	Matrices	Matrices computations	Analysis (historical occurrence)	Ranking	
Agarwal and Cagan [1998]	Shape-grammar	Shape production rules	-	-	The generation is performed via a manual application of production rules
Rai [2011]	Modelica model	Graph production rules	Simulation	-	Safety structural requirements are encoded in rules
Kerzhner and Paredis [2009]	SysML model	Graph production rules + Evolutionary Programming mechanisms	Structural properties*	Evolutionary programing	*such as number of pumps in a hydraulic circuit
Helms and Shea [2009]	Graph	Graph production rules	-	-	

Tab. 2.1: CDS methods classification in the CDS framework (1 of 2)

Reference	Representation	Generation	Evaluation	Guidance	Comments
Kurtoglu and Campbell [2009]	Graphs	Graph production rules	-	-	Production rules transform function structures into component structures and are deduced from existing products
Holley [2011]	Matrices	Combinations	Quality Function Deployment	-	
Aleti et al. [2010]	AADL model + Matrices	Evolutionary mechanisms	Network properties (bandwidth, reliability...)	Pareto selection or Fitness selection	
Seo et al. [2003]	Bond-graphs	Graph production rules + Genetic Programming mechanisms	Causality analysis + State model	Fitness selection	
Bolognini et al. [2007]	Connected Node System	Burst algorithm	FEM analysis	Pareto selection	
Alber and Rudolph [2004]	Trees	Graph production rules	FEM analysis	-	

Tab. 2.2: CDS methods classification in the CDS framework (2 of 2)

CHAPTER 3

THESIS SCIENTIFIC OBJECTIVES

Given the review of existing CDS method and comparing them to the industrial needs, some objectives of the research works can be formulated.

Computational methods proved to be efficient in supporting the design activities. But no method fits the current industrial needs (see p. 31). The main objective of this thesis will be to design a method that covers the industrial needs. For this, building blocks of existing methods may be reused in an original manner. Some new techniques may be developed in case no existing mean is available to cover a need in the method.

The design method shall:

Objective 1 permit to generate design alternatives given a functional architecture of the system and a library of component types.

Objective 2 permit to identify best design alternatives.

Objective 3 be able to work on large design-space where the evaluation of every solutions is not possible.

Objective 4 not require the explicit description of compatibilities between the elements of the problem (functions and components).

Objective 5 be in line with the Model-Based Systems Engineering initiative.

Part IV

CONTRIBUTIONS

CHAPTER 1

OVERVIEW

Given the industrial and scientific objectives of the research works, a need for a new method supporting the architecture design process was highlighted. In this part, this new method will be described.

Making reference to the framework proposed by Cagan et al. [2005], we can give the composition of our CDS method :

Representation Model

Generation Tree search (Breadth-First Search) + Genetic operators

Evaluation Evaluation modules

Guidance Pareto selection or Preference-based selection

The following chapter will go into details for each step. Arguments for the choice of the method characteristics are given.

CHAPTER 2

REPRESENTATION

2.1 Introduction

As seen in Part III many CDS methods rely on matrices or graphs to represent the problem and its solutions. However, these representations are limited in their representation power i.e. they can hardly enclose numerous and heterogeneous information on the elements of the problem. On the other hand, system models permit to represent the problem along several views and to store complex data. Also, they present the advantage of being reusable, easily maintainable and to enhance communications between actors of the design process when associated to graphical notations.

2.2 Model and meta-model

A model is a representation of a real artifact built for given purposes. In our case, the artifacts to model are a design problem and its solutions. The purpose of the model is to permit the synthesis of solutions to the design problem and their analysis. Nevertheless, as a text needs to be associated to a language to be understood, a model needs its "language". It is the meta-model (or data-model).

A meta-model defines the concepts that will be manipulated in the model. These concepts must be defined explicitly and non-ambiguously to ensure the same comprehension between different users of the meta-model.

Relations between the different concepts must also be defined. As concepts, relations must be defined to guarantee their correct understanding. For instance, the UML meta-model includes a composition relation between two classes which is defined in the language specification [OMG, 2011]

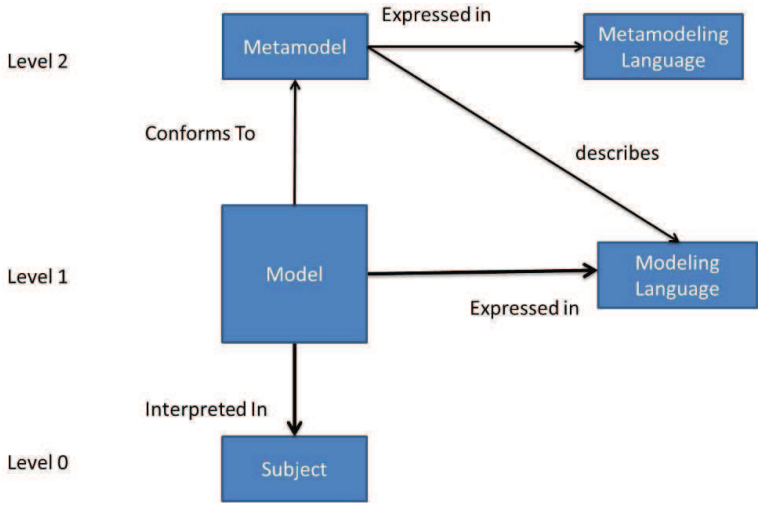


Fig. 2.1: Meta-model, model and language (from [OMG, 2012])

2.3 Meta-model presentation

We analyzed different design problems coming from different industries and we looked at the concepts manipulated in systems engineering and operational research literatures. From this analysis, a meta-model that gathers all the necessary concepts for our method was built. This meta-model is made of concepts related to the decision-making process and to optimization and concepts related to the representation of system architectures. The first ones permit to describe the worth of an architecture (criteria, objectives, constraints..) while the second ones permit to describe the functional architecture, the components, the system and its environment.

This meta-model is represented in Figure 2.2.

Model The model concept is used to organize the structure of the model. It comprises objectives, criteria, constraints, populations, types as well as the definition of the system.

2.3.1 The decision concepts

Alternative An alternative is a potential solution to a decision. This concept is used to make the link between the architectural aspects and the decisional aspects. An alternative is linked to an architecture.

Population A population is a set of alternatives. This concept is useful when an evolutionary optimization process is used.

Criterion A criterion is an aspect that will be considered during selection or decision-making.

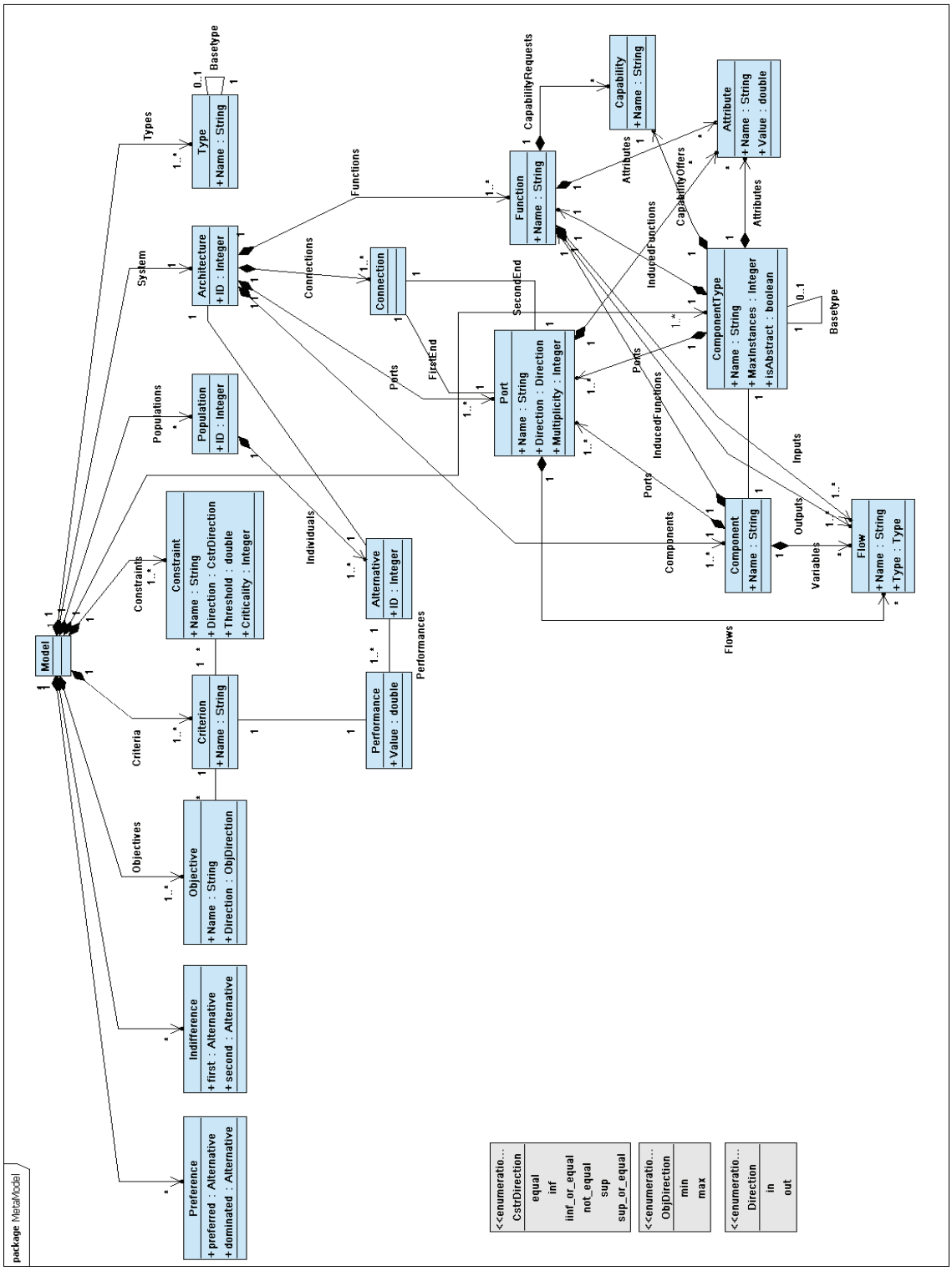


Fig. 2.2: Meta-model overview

Performance A performance is a value representing the evaluation of an alternative on a criterion. Thus, attributes of an alternative such as mass or cost are considered as performances of the alternative.

Objective An objective is a desired characteristic of alternatives. Objectives are considered to define the worth of alternatives. It is related to a criteria and comprises an optimization direction defining the direction of improvement desired by the decision-makers (minimize/maximize).

Constraint A constraint is a mandatory characteristic of alternatives. Constraints are considered to define the acceptability of alternatives. Generally speaking, a constraint is defined as a boolean condition involving performances of the alternative. In our meta-model, a constraint is defined by a criterion, a conditional operator ($>, \geq, =, \leq, <, \neq$) and a target value. If the performance of an alternative on the defined criterion does not fulfill the condition defined by the operator and the target (e.g. "$>$10") , then the constraint is violated by the alternative. Depending on the selection process and the context, it may be useful to give different importance to constraints. This can be done by the use of a criticality index (integer).

Preference A preference is an information given by a stakeholder (or a group of stakeholders) stating that, in its opinion, an alternative is at least as good as another. A preference is composed of a preferred alternative and a dominated alternative. This concept is used in preference-based selection algorithms.

Indifference An indifference is an information given by a stakeholder (or a group of stakeholders) stating that two alternatives are equally preferred. A preference is composed of a preferred alternative and a dominated alternative. This concept is used in preference-based selection algorithms.

2.3.2 The architecture concepts

Architecture An architecture is the organization of the system. It comprises components connected together by connections, functions of the system and interfaces with the system environment (ports).

Type Types represent a classification of flows exchanged by functions or components. A generic classification of types in engineering was proposed by Magee and De Weck [2004] : matter, energy, information and value. Sub-types of these basic types can be created (e.g. *electrical energy*).

Flow A flow represents an exchange of matter, energy, information or value between two entities (functions or components).

Port Ports are points of interaction between components and/or systems. They permit the exchange of flows. Ports are typed so that only compatible flows (i.e. flows having the

same type or a child of the port type) can flow through them. A port has a direction (in or out) that constraints the way flows can go in this port. A *multiplicity* attribute states the maximum number of connections that can go from/to a port.

Component Type A component type is an abstract class that can be instantiated into an architecture, thus creating components. Component types can be declared as *abstract*. In this case, the component type is only used to simplify the description of children component types (derived from it) thanks to the inheritance mechanism and cannot be instantiated. For instance, an abstract "generic motor" component type can be described and "DC motor" and "AC motor" component types will be declared by inheritance of the "generic motor" component type. A constraint can be put on the maximum number of instances of a component type that can be put in an architecture through the *max_instances* attribute. Its default value is infinite.

Component A component is the concrete realization of a component type in an architecture. It can exchange flows with other components or with the system environment through its ports. Functions can be allocated to it. A component can embed variables. Variables are flows that are "known" by the component i.e. natively present in the component or acquired through ports or by realization of a function.

Function A function is a task that transforms input flows into output flows. A function can be realized by (allocated to) a single component or by a set of components. A function can require capabilities from the components that realize it.

Capability A capability represents a resource required by a function to be performed. In order to realize this function, the set of components allocated to the function must embed the required capabilities of the function. A capability is only qualitative and is represented by its name.

Attribute An attribute is a piece of information that can be used by the algorithms used to synthesize and optimize architectures. They can be used, for instance, to compute the performances of an architecture (e.g. mass) or to check compatibility between elements of the problem (functions and components). An attribute is characterized by its name and its value (e.g. name = Mass, value = 15).

The advantage of using a meta-model is that it can be used by software applications to work on models. The model implements concepts defined by the meta-model. As the application knows the meta-model, it is capable of interpreting the model. The meta-model will notably be used by an algorithm for the generation of design alternatives.

CHAPTER 3

GENERATION

The design-space of complex system architectures is generally huge. Many functions can be realized in many different ways i.e. allocated to different components. For this reason, generating all solutions of the problem is unfeasible.

In order to explore the design-space in a smart way, the CDS method uses evolutionary mechanisms. Evolutionary methods are based on a selection-evolution-evaluation scheme which permits to iteratively discover solutions with increasing value. They proved to be robust ways to solve non-linear multi-objective optimization problems with discrete decision variables.

Evolutionary algorithms consist in the following procedure :

Algorithm 1 Generic genetic algorithm

```
Create an initial population P_0
Evaluate the performances of individuals
Select the best individuals {See Selection}
while Stop criterion is not met do
    Create a new population based on selected individuals {See Evolution}
    Evaluate the performances of individuals
    Select the best individuals
end while
```

Generally, the first population of an evolutionary algorithm is generated randomly i.e. individuals are created by selecting randomly a value in the domain of each solution characteristic. However, proceeding so, there is a negligible probability that the initial solutions are acceptable. Thus, the first iterations are dedicated to the discovery of feasible solutions. This causes to spend computational time.

In order to increase the performances of the evolutionary approach, the initial population of solutions is created thanks to a synthesis algorithm (see Section 3.1). This permits to start the process with solutions already acceptable with respect to generic rules (See

Section 3.2). In next iterations, genetic operators are applied to the current population to obtain new solutions (see Section 3.3).

3.1 Architecture synthesis

3.1.1 Overview

The goal of the synthesis algorithm is to build design alternatives based on the information of the model (system functions, system interfaces and component type library) and on some rules. In order to generate architectures, the algorithm will add components to an initially empty architecture and will allocate functions to them. For each function, the algorithm will try to assembly components in order to form chains that will realize the function. When trying to allocate functions, the algorithm considers component types of the library but also components already present in the architecture. This permits to allocate several functions to a same component. Functions are treated in a backward manner i.e. from output functions (those which do not supply other functions) to input ones. The pseudo code of the algorithm is presented in 2.

Algorithm 2 Synthesis algorithm

1: Create a new architecture
2: Put architecture functions in a stack F_{to_treat} and order them
3: **while** $F_{to_treat} \neq \emptyset$ **do**
4: Take (and remove) the first function f of the stack
5: Search sinks and sources for f
6: Search possible chains for f
7: Choose a chain randomly
8: Implement the chosen chain
9: **end while**

3.1.2 Search algorithm for component chains

During step 6 , the algorithm searches possible component chains for the function F. To do so, the algorithm will perform a breadth-first search (BFS).

It will first search chains of size 1 (one component) and test them, then of size 2 (two components), etc...

The depth of the search is dynamically limited. When a possible chain of size N is found, the algorithm will search chains of size up to $N + \delta$ where δ is a user-defined parameter of the algorithm. This parameter should be used with care as it can cut a part

of the design-space if it is set to a low value. The introduction of the δ parameter was driven by 2 facts :

1. the objective of the search is not to find the shortest chain but a chain capable of realizing a function. The search must be continued once a shortest chain has been found.
2. the branches of the solution tree can be infinite and the search would be infinite without a stop criterion.

Also, in order to limit the search, the algorithm only explore chains starting with a component connectible to sources and ending with a component connectible to the sink.

Algorithm 3 Chain search algorithm

```
1: s ← 1
2: while No chain found and limit size not exceeded do
3:     Search chains of size s
4:     s ← s + 1
5: end while
6: for t = stos + δ do
7:     Search chains of size t
8: end for
```

3.1.3 Introduction of a heuristic for reduction of complexity

In order to favor the reuse of existing components over the creation of new ones, a parameter called *Reuse probability* $\in [0; 1]$ is introduced in the synthesis algorithm. This parameter is used during the breadth-first search of chains. At this time, the algorithm collects component types and components into a list. During this collection, component types are placed at the end of the list with the probability *Reuse probability*. If the parameter is set to 0, then the order of the list is random.

By tuning this parameter, the complexity of synthesized architectures can be controlled. This shall permit to improve the convergence of the optimization algorithm on complexity related objectives but shall also have impacts on some safety related criteria as it may create common nodes between functions.

3.1.4 Implementation of chains

At the end of the search of component chain, the selected chain is represented as a list of component types from the library and of components present in the architecture.

Components represent actual components that can be reused whereas component types represent future instances that will be added to the architecture during ht implementation.

The implementation phase consists in modifying the architecture in order to include the selected chain and to connect it to selected sinks and sources. It is performed in the following way :

Algorithm 4 Implementation algorithm

1: **for** $i = 1 \rightarrow chain_size$ **do**
2: **if** C_i is a component **then**
3: Add a components to the architecture $\rightarrow CI_i$
4: **else** $\{C_i$ is a component$\}$
5: $C_i \rightarrow CI_i$
6: **end if**
7: Allocate the function to CI_i
8: **if** $i \neq 1$ **then**
9: Connect CI_i to CI_{i-1}
10: **end if**
11: **end for**
12: **if** sources $\neq \emptyset$ **then**
13: Connect sources to CI_1
14: Add input flows to corresponding ports
15: **end if**
16: **if** $i = chain_size$ **then**
17: Connect sinks to CI_i
18: Add output flows to corresponding ports
19: **end if**
20: Add function input flows to CI_1

In components, variables are used to model the "awareness" of a flow by the component. The first component of a chain will be aware of the inputs of a function which can be

- received from other components,
- received from the system environment,
- due to the realization of a function by the component itself .

Following the same principle, the last component of a chain realizing a function will be aware of the outputs of the function and will have it in its variables.

This information is used by the algorithm to find sinks and sources and notably permits the realization of a chain of functions by the same component.

For instance, in the architecture presented in Figure 3.10 (77), $C1_1$ owns the vari-

ables $f1$ and $f2$ and $C3_1$ owns the variables $f3, f4$ and $f5$ as it realizes functions $F2$ and $F3$. During the search of a chain to realize $F3$, $C3_1$ will be used as a sink since the flow $f4$ is an output of the function and is contained in its variables.

As seen before, a weakness of state-of-the-art methods lies in the fact that compatibility between components has to be stated explicitly or is based on existing products. In order to generate coherent architectures, the algorithm uses implicit rules to make connections between component instances and to allocate functions to components.

3.2 Rules

When looking for architecture concepts, designers consider both functional and technological aspects. In order to generate coherent architectures, the formal design synthesis method must consider both aspects and formalize the associated constraints under the form of rules.

3.2.1 Viability

On the physical side, designer only consider assemblies of components that can be connected together and that can be connected to the environment if flows must go in or out of the system. An architecture is said to be viable if all its connections are coherent. A connection is considered as coherent if the ports that are connected are compatible.

Port compatibility

The compatibility between two ports is defined by the following sub-rules:

Direction The two connected ports must have opposite direction. An input port can only be connected to an output port and vice-versa.

Type The two connected ports must have a compatible type. It can be the same type or a parent type.

Multiplicity The two connected ports must have a strictly positive remaining multiplicity rm. The remaining multiplicity of port p is

$$rm_p = M_p - c_p \tag{3.1}$$

with M_p the initial multiplicity of the port and c_p the number of connections connected to the port.

Other The rule can be completed depending on the problem. The added sub-rule can use the attributes defined in the ports or in the components. For instance a "male/female" sub-rule can be added. This rule checks that if a "male/female" attribute is defined on each connected port, their value is opposite on each port.

The viability rule

The viability rule checks if the connections in a chain of components are coherent or not. In other words, it checks if connected components are compatible. For this, the rule checks, for each pair of connected components, if it exists ports on each component that can be connected together.

3.2.2 Validity

On the functional side, designers look at assemblies of components that can realize the system functions and that can make function inputs and outputs flow in,out or into the system. An architecture is considered as valid if all its functions are fulfilled by chains of components capable of realizing their allocated functions. A chain of components is capable of realizing (valid for) a function if

$$C_f \subset \bigcup_{c \in chain} C_c \tag{3.2}$$

where C_f is the set of required capabilities of the function f and C_c is the set of offered capabilities of the component c.

Furthermore, functions are realized by taking flows from some sources, transforming them and sending the resulting flow(s) to sinks. Sources and sinks can be located at the interfaces of the system i.e. input (respectively outputs) are gathered from (respectively sent to) the environment of the system. Consequently, the viability of a chain is also conditioned by its capacity to create (or to contribute to the creation of) a flow circulation between inputs and outputs of the system.

These two aspects are formalized by the viability rule.

3.2.3 Other rules

To allow flexibility in the design rules, other rules can be defined by designers. These user-defined rules will be considered by the algorithm during architecture synthesis in complement to the viability and validity rules. These extensions must exploit the information contained in the model.

For instance, we define a "minimal chain" rule. This rule permits to limit the complexity of architectures and the size of the design space. A chain is considered as minimal if and only if no sub-chain (using the same components) of it is viable and valid. This permits to eliminate chains with components that do not participate to the realization of a function or to the connectivity with other components. Thus, only chains with the minimum size, given a set of components, will be authorized to be inserted in architectures.

3.2.4 Example : Architecture synthesis

In order to illustrate the method in details, we propose a simple theoretical problem. This problem does not permit to illustrate all cases encountered by the algorithm but most common ones.

The system must realize 3 functions ($F1, F2$ and $F3$),$F3$ being a supplier of $F2$. Functions have inputs and outputs flows fx of types Tx. Furthermore, the function $F2$ requires a capability c.

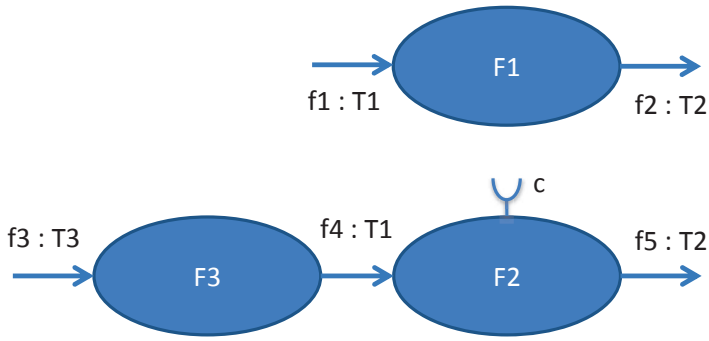


Fig. 3.1: Functional architecture

Four component types ($C1, C2, C3$ and $C4$) are available to form the system architecture. Each component type has an input and an output port. The type and multiplicity (between parenthesis) of each port is indicated in 3.2. Component types $C2$ and $C3$ have a capability c. The component type $C3$ has an induced function $FI3$.

The system has three interfaces (system ports) that permit to exchange flows with the environment.

Based on the problem description, we will illustrate the architecture synthesis process. Let us suppose that the parameter δ is set to 1.

First, let us consider that $F1$ is treated as it is an output function[1]. $F2$ could also

1. An output function is a function that has at least one output flow not supplied to another function

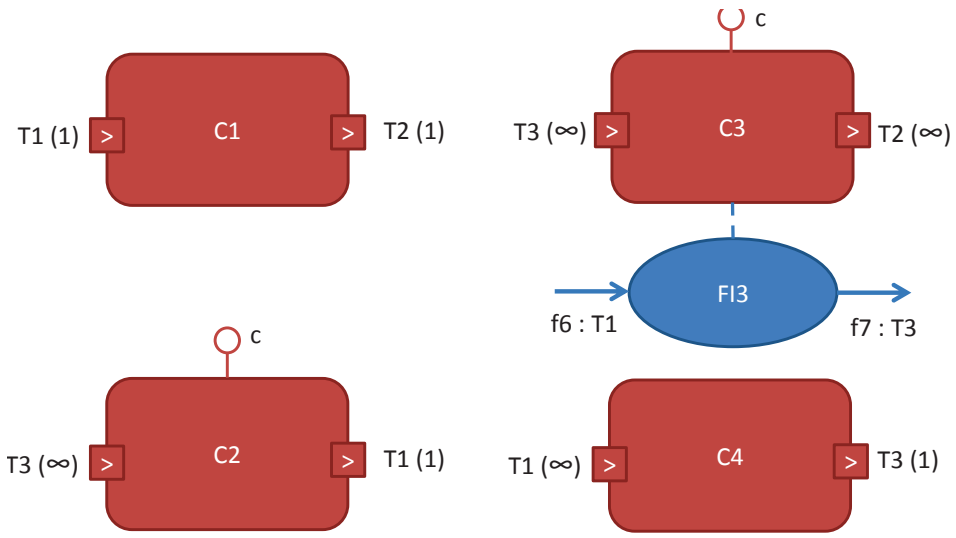


Fig. 3.2: Component types

Fig. 3.3: System

be treated first. This would have no effect on the synthesis problem as the combinatorics would remain the same.

As $F1$ is both an input[2] and an output function, the algorithm looks for sinks and sources in architecture ports. For $F1$, the unique solution is $p1$ as source and $p3$ as sink.

The algorithm starts searching chains of order 1. $C1$ can realize the function as it is connectible to $p1$ and $p2$. Then the algorithm searches chains of order 2 and finds that the chain $C3 \rightarrow C4$ is also a possible chain.

The two chains have the same probability to be selected and can be implemented (Figure 3.5).

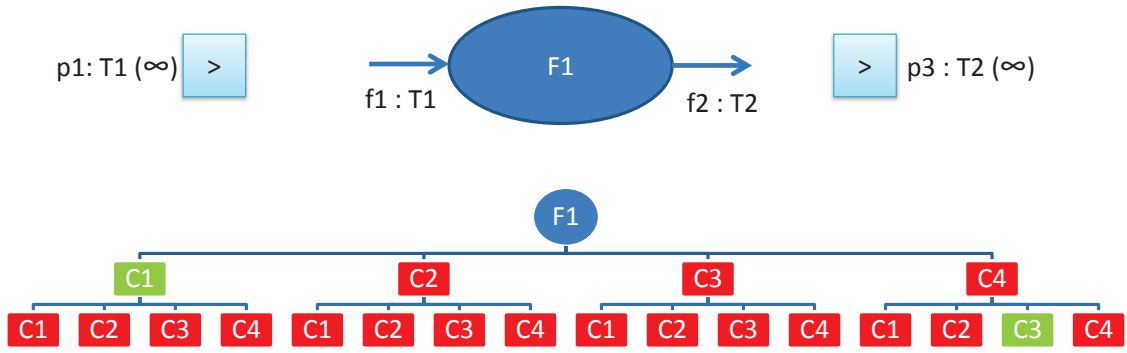
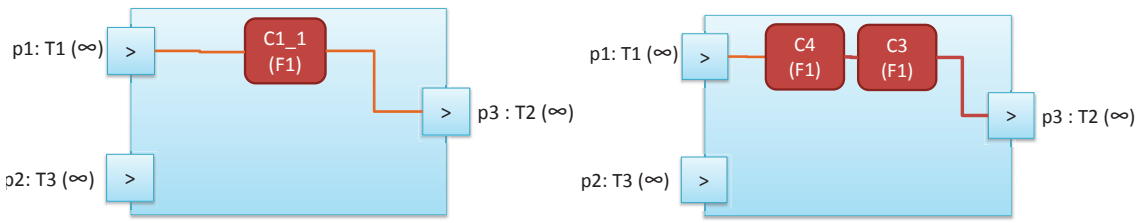


Fig. 3.4: Treatment of $F1$



(a) Choice of $C1$ *(b)* Choice of $C3 \rightarrow C4$

Fig. 3.5: Architectures after $F1$ treatment for the two possible chains

Let us suppose that $C1$ is selected. The chain must be implemented. An instance $C1_1$ of the component type $C1$ is added to the architecture and $F1$ is allocated to it. Two connections are added between $p1$ and the input port of $C1_1$ and between $p2$ and the output port of $C1_1$. $f1$ and $f2$ are added to $C1_1$ variables. $f1$ is added to the flows of the input port of $C1_1$ and $f2$ is added to the flows of the output port of $C1_1$.

2. An input function is a function that has at least one input flow not supplied by another function

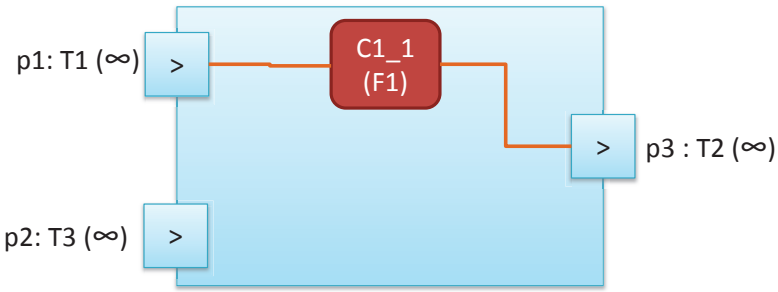


Fig. 3.6: Architecture after $F1$ treatment

The second chain to be treated is $F2$. $F2$ is an output function so a sink must be looked for in architecture ports. $p3$ is the only available port and is compatible with the output flow $f5$ as they are both of type $T2$.

The chains that are eligible for realizing $F2$ must be connectible to the sink $p3$ but must also comprise the capability c. The component type $C3$ is able to realize the function alone since it fulfills these conditions. For the same reason, the chain $C2 \rightarrow C1$ is eligible. The chains ending with $C1_1$ cannot realize the function because $C1_1$ cannot realize the function by itself and cannot be connected to another component as its input port has reached is maximum multiplicity (1). The two architectures corresponding to each option are presented in Figure 3.5.

The chain $C3 \rightarrow C4$ could realize the function. Nevertheless, if the "minimal chain" rule is enabled, this chain is only an extension of the chain $C3$ which can realize itself the function. In this case (which we will consider here), the chain $C3 \rightarrow C4$ is ignored.

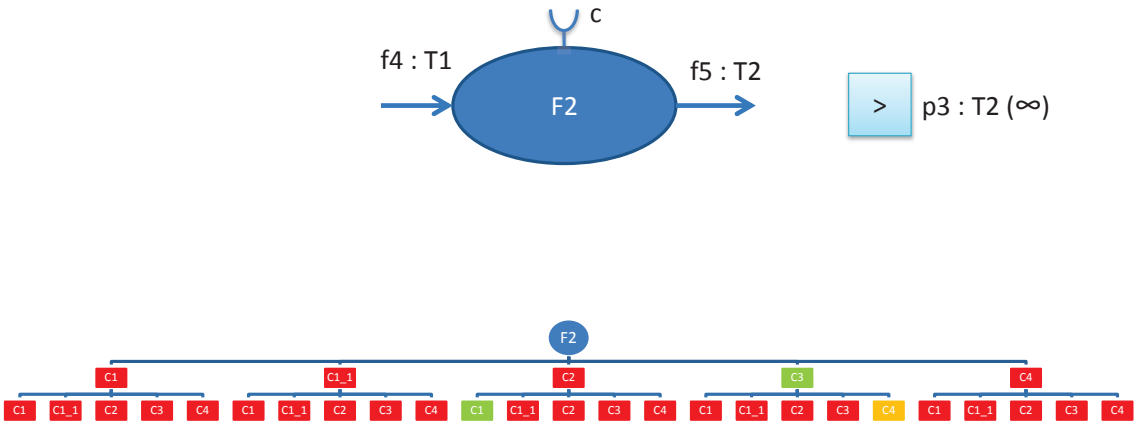
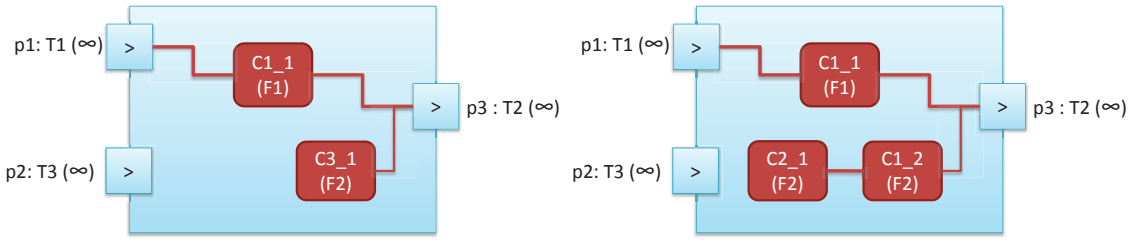


Fig. 3.7: Treatment of $F2$

Note that, if the *Reuse probability* would be set to 1, chains reusing $C1_1$ would



(a) Choice of $C3$ (b) Choice of $C2 \rightarrow C1$

Fig. 3.8: Architectures after $F2$ treatment for the two possible chains

be checked before other chains. In the current case, since $C1_1$ does not belong to any possible chain, there is no impact.

The two options have the same probability to be selected. Let us suppose that $C3$ is selected. In this case, the chain is implemented similarly to the previous implementation ($C1$ for $F1$). However, as $C3$ owns an induced function $FI3$, an instance $FI3_1$ of this function is added to the system functions. The list of functions to treat is $[F3; FI3_1]$.

$F3$ is now treated. The source for the input flow $f3$ is $p2$. The sink for the output flow $f4$ is $C3_1$ as its variables includes this flow. For this function, the component $C3_1$ is eligible since it is the sink of the function and it can be connected to $p2$. The chain $C2 \rightarrow C4$ is also a possible chain to realize $F2$.

Let us suppose that $C3_1$ is chosen to realize $F3$. The architecture after implementation is presented in Figure 3.9.

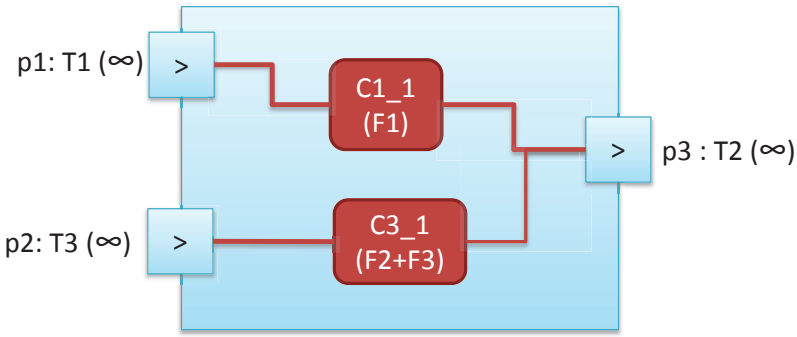


Fig. 3.9: Architecture after treatment of $F3$

The last function to allocate is $FI3_1$. The source is $p1$ and the sink is the input port of $C3_1$ (as it includes $f7_1$ in its flows). The only solution to realize the function is $C4$. Once the implementation of the chain is performed, the synthesis of the architecture is complete (Figure 3.10).

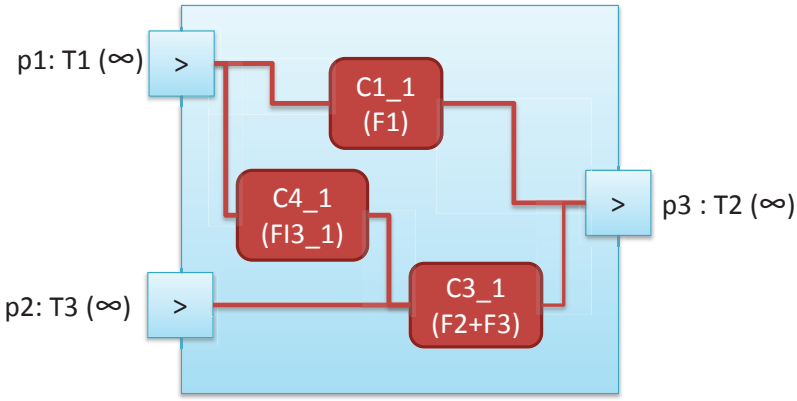


Fig. 3.10: Final architecture

Note that if, during the process, the breadth of the chain search exceeds a given limit or if no sink/source is found, the synthesis process is restarted.

On this simple problem, the synthesis algorithm is able to find different solutions (6).

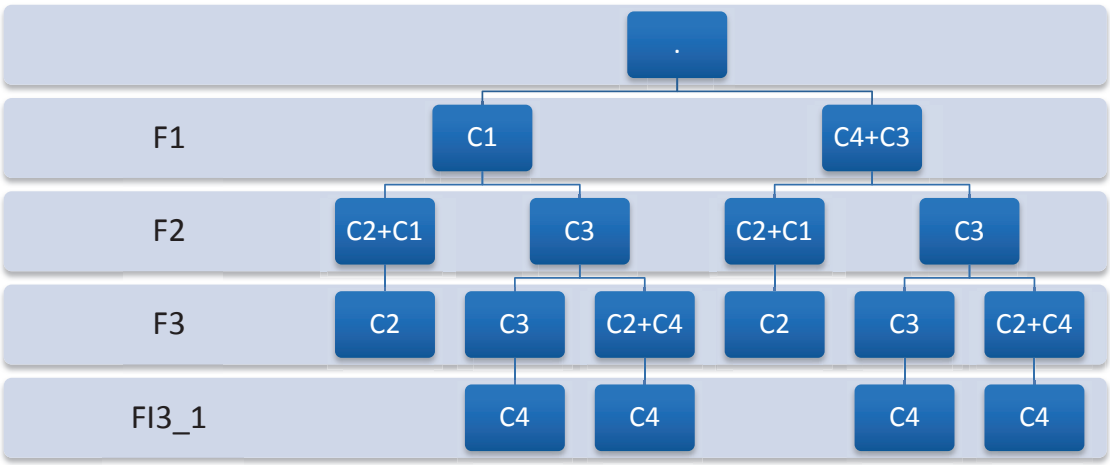


Fig. 3.11: Different possibly generated architectures

The synthesis method presented above permits to generate potential solutions to the design problem i.e. the synthesis of a physical architecture able to fulfill the system functions. The combinatorics of the problem may be huge depending on the considered func-

tions, components and constraints(up to 10^{21} on some problems). Consequently, it is often prohibitive to generate all solutions of the problem (full-factorial study). In this context, it is interesting to use an optimization method that will perform a guided exploration of the design space.

3.3 Evolution

As stated before, for complex systems, it is often impossible to generate every solutions of the design-space. In this case, an optimization method must be used. We selected an evolutionary approach adapted from genetic algorithms since :

- it permits to obtain a global optimum,
- it can work on multi-objective problems,
- it is problem independent.

Genetic algorithms are mainly based on evolution mechanisms applied iteratively to a populations of solutions. These evolution mechanisms are often called genetic operators. Holland [1975], the "father" of genetic algorithms, defines two genetic operators : mutation and crossover. Working with classical genetic algorithms using a chromosome, several types of genetic operators are available (single point, 2-points, uniform crossover / bit inversion, order changing mutations). Nevertheless, using these classical operators, there is no guarantee that the created individual will respect the constraints i.e. will be viable and valid in the case of architectures. In order to generate only viable and valid architectures, we propose to create genetic operators :

- adapted to our description of the problem (system models),
- that guarantee the satisfaction of the viability and validity rules.

3.3.1 Mutation

Mutation is a genetic operator that generates a child based on a single parent by modifying its characteristics. In classical genetic algorithms, mutation is realized by changing some genes of the parent.

In our context, we define several types of mutation operators : simple mutation, redundancy creation and redundancy removal.

This operators are very similar to the graph transformation rules used in graph-grammar based methods. A part of the initial architecture is replaced to form a new architecture. Nevertheless, there is no need to define explicitly every possible mutation rule.

Simple mutation

Simple mutation is realized by reviewing the realization of a functions.

Algorithm 5 Simple mutation algorithm

```
Select an output function F and the direct and indirect supplier functions {F_s} of F
Select a network of components realizing F and {F_s}
Remove allocation of F and {F_s} from the chain
Remove unused components , their associated induced functions and the chains realizing them
Remove unused connections
for F and {F_s} do
    Search possible chains {Same as initialization}
    Select a chain
    Implement the chain {Same as initialization}
end for
if Child ≠ parent then
    return Child
else
    return ∅ (null) {The mutation operation fails}
end if
```

In case the created child is identical to its parent (i.e. if the removed network is identical to the added one), the operator returns nothing and has no effect on the new population (no individual is added).

In the same manner as for the synthesis algorithm, the *Reuse probability* parameter is considered to favor the reuse of existing components.

Example : Mutation

Let us consider that the "simple mutation" operator is applied to the final architecture presented in Figure 3.10. The operator will first select a function ($F3$) of the architecture and will remove the corresponding chain.

Then, a chain to realize the removed function will be searched. The search considers the sources ($p2$) and sinks ($p3$) used by the removed chain. As seen before, the chain $C2 \rightarrow C1$ is able to realize $F2$ and $F3$ and can be implemented (Figure 3.13).

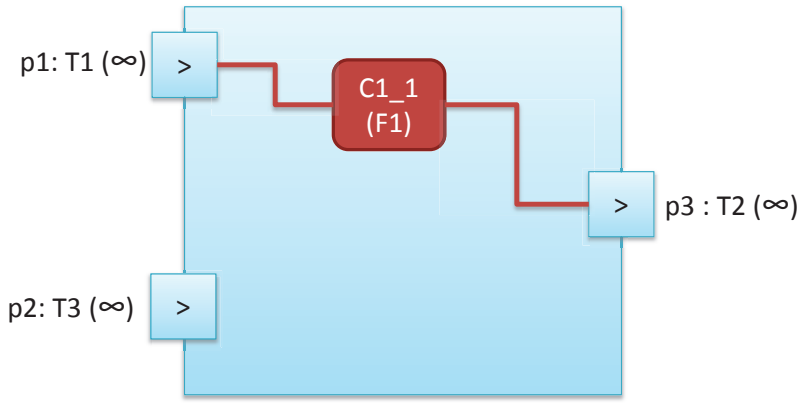


Fig. 3.12: Architecture after the removal of chains ($F2$,$F3$ and $FI3_1$)

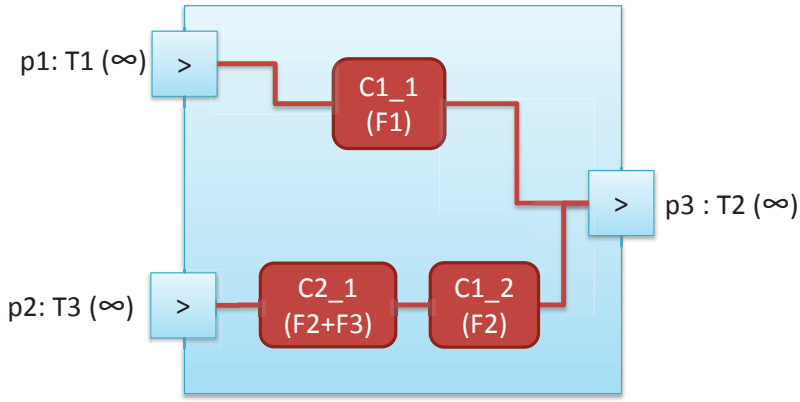


Fig. 3.13: Architecture after simple mutation

Redundancy creation

Redundancy creation permits to create a new path realizing a function in parallel to others paths realizing the same function. Adding redundancies is particularly important for RAMS-related criteria as it permits to increase the reliability and availability of functions by providing alternative means to realize them. The redundancy creation is realized in the same way as simple mutation, except that no chain is removed. The algorithm selects a function, search possible chains to realize it and adds the chain to the new architecture.

3.3.2 Crossover

The crossover operator is a two-parents operator. It mixes the characteristics of two parents (A and B) in order to obtain two children (C and D). The crossover must operate on common characteristics of the 2 parents. In our case, the only common characteristics of parents are the system functions (architecture functions except induced functions). The operator analyses the way these functions are realized in parents and reproduce similar chains (using same component types) in C or D.

The crossing scheme of the crossover operator determines which functions must be reproduced (A to C, B to D) and which must be exchanged (A to D, B to C). We selected a random crossing scheme. When the operator is applied to parents A and B, functions to reproduce are selected randomly and remaining functions are exchanged.

Algorithm 6 Crossover algorithm

```
Determine crossover scheme
for all Output functions to be reproduced do
    Clone the chain from A to C
    Clone the chain from B to D
end for
for all Output functions to be exchanged do
    Exchange the chain from A to D
    Exchange the chain from B to C
end for
```

The reproduction mechanism analyses chains realizing the function in the parent architecture and reproduces it exactly in the child architecture. The mechanism builds a mapping between parent components and child components. Consequently, if two reproduced functions are allocated to a component in the parent architecture, the functions will also be both realized by the corresponding component in the child architecture.

The exchange mechanism is slightly different as it considers the components already present in the child after the application of reproduction. Thus, components can be reused or new instances can be created. In the same manner as for the reproduction mechanism, the exchange mechanism produces a mapping between parent and child components.

Nevertheless, it may be the case where, because of constraints on the problem, the exchange mechanism cannot respect the mapping or cannot even respect the types of components used to realize a function. For this reason, the exchange mechanism is given more flexibility by allowing him to slightly diverge from the nature of the parent chain. For this, the exchange mechanism uses the same breadth-first search algorithm as for initialization, but places:

1. The component from the mapping (if existing) at the first position
2. The components of the same type and the component type in the following positions
3. All other components and component types at the end.

This way, the algorithm will first check if the chain issued from the mapping satisfies the rules. If not, it will try to maximize the use of the mapping components or, as a second criterion, to maximize the use of same types of components in the chain.

3.3.3 Example : Crossover

Let us consider that the "crossover" operator is applied to the two sample architectures. The chain realizing $F1$ are cloned from parent1 to child1 and from parent2 to child2. The chain realizing $F2$ and $F3$ are exchanged (parent1 to child2 and parent2 to child1)

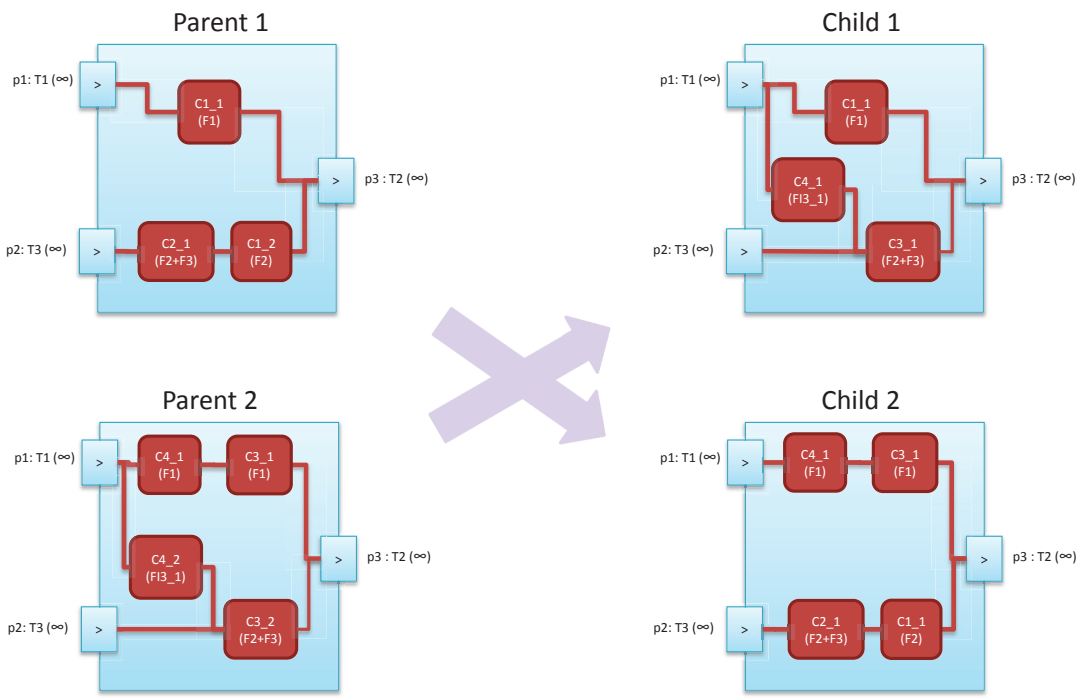


Fig. 3.14: Crossover

CHAPTER 4

EVALUATION

As the criteria on which decisions are made vary from one design problem to another, the solution proposed to evaluate design alternatives is generic and must allow to evaluate different kinds of performances.

We use the word "Evaluation module" to define algorithms, programs and softwares that compute the performances of an architecture along a criterion. These modules gather information from the model and aggregate them to compute the performance. Used information can be structural and/or quantitative (attributes).

Evaluation modules can be custom computer programs, developed for the current design problem, or can be commercial analysis or simulation tools. Custom programs are developed specifically for the meta-model and are thus able to directly interpret models.

At the contrary, using commercial tools require to transform the model into the formalism of the tool. For this, model-to-model or model-to-text transformation languages and associated model transformation tools (ATL [Jouault et al., 2008], Acceleo [Obeo, 2012]...) can be used.

Setting up the model transformation represents an effort but is compensated by the fact that the analysis or simulation algorithms are already developed. Commercial tools shall be used only if:

- the effort for setting up the model transformation is weaker than developing a custom evaluation module
- and the computational cost for evaluation is not prohibitive.

Commercial tools must also if setting up a custom module requires knowledge that is not available.

Often, the use of a custom program is generally made possible by making some simplification hypothesis. For instance, the electrical consumption of a system shall be assessed by using simulation on a representative scenario. Doing so would provide a precise measurement of the consumption of the architecture but would require a model-transformation to

be able to use a simulation tool such as Simulink (Figure 4.1) or Modelica platforms. Alternatively, the evaluation module can compute the mean of the average consumption of each component. This assumes that every components of the architecture work at their mean consumption. This mode of evaluation introduces more uncertainty than the simulation mode but may be sufficient in a first stage of design optimization.

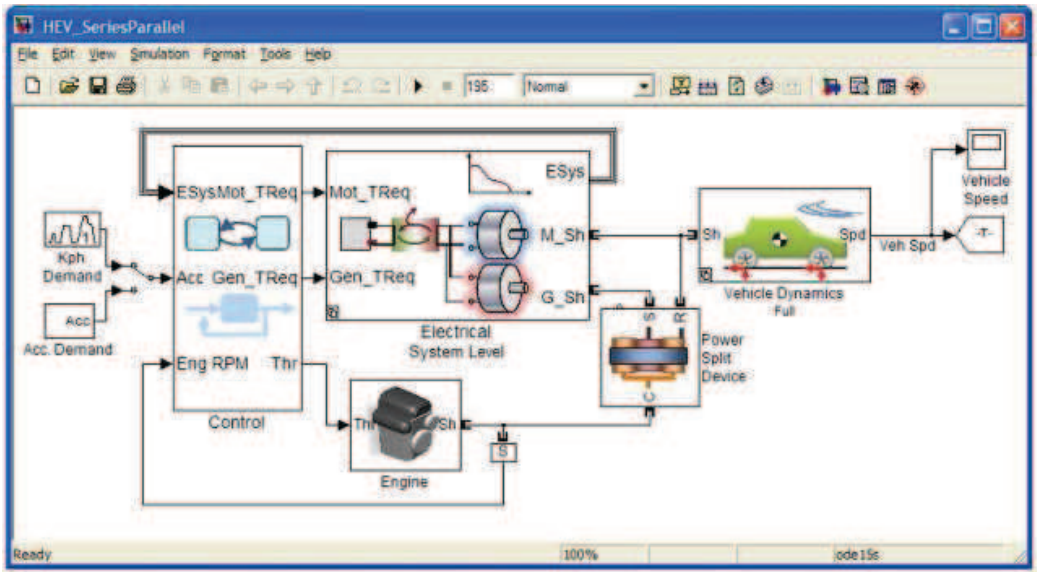


Fig. 4.1: A Simulink model

A first strategy for the evaluation process consists in requesting evaluations of every criteria systematically after the generation stage. Nevertheless, as it will be exposed in Chapter 5, the selection process only looks at objectives when all constraints are fulfilled. In order to improve the performances of the overall process, evaluations can be executed on demand i.e. the performance on one criterion is computed only when the performance is requested by the selector.

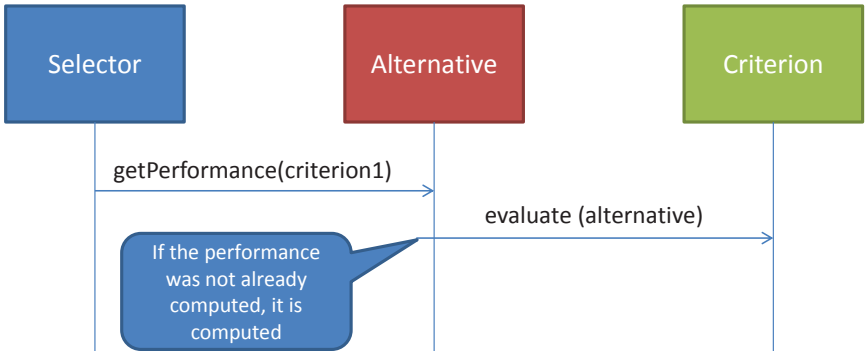


Fig. 4.2: On-demand evaluations of performances

CHAPTER 5

SELECTION

The selection phase consists in selecting the best individuals of a population in order to create a new population. In a multi-objective context, the definition of the worth of an individual is not trivial. As stated in Part III, three approaches can be adopted to work with multi-objective problems : fitness function use,a Pareto approach or a preference-based approach. The fitness function apporach will not be considered because of the difficulty to set up the fitness function and the sensitivity of the results with respect to the fitness function choice.

The two remaining approaches have different modes of deployment : the Pareto approach does not require intervention from the user but gives generic solutions while preference-based approaches require user interactions to select solutions that have high value in the eyes of the user. Although the use of a preference-based approach is preferable because it faster the convergence of the optimization process toward valuable solutions, the requirement of having the user available all along the process is often too demanding to be implemented in an industrial context. For these reasons, the two approaches are made available to the method users that will have to choose one of them depending on the context.

5.1 Pareto approach

The Pareto approach defines a dominance relation between two alternatives (or individuals) based on their performances:

A solution x_1 dominates a solution x_2 ($x_1 \succ x_2$) if $\forall i g_i(x_1) \geq g_i(x_2)$ and $\exists i$ such as $g_i(x_1) > g_i(x_2)$.

To implement this approach, we selected the famous NSGA-II algorithm [Deb et al., 2000]. This algorithm is interesting as it is elitist (i.e. explicitly guarantees the survival of best individuals) and comprises an explicit diversity preserving mechanism (in the objective

space). The algorithm first sorts the solutions with respect to their Pareto dominance, forming Pareto fronts F_i:

$$\text{If } x_1 \succ x_2 \text{ then } i < j. \text{ with } x_1 \subset F_i \text{ and } x_2 \subset F_j$$

The non-dominated solutions in S (whole population) are on the first Pareto front F_0. The non-dominated solutions in $S - F_0$ are on the second Pareto front F_1, etc...

In order to obtain a complete ranking, a crowding distance is used to sort individuals belonging to the same front. This crowding distance permits to favor diversity of solutions.

Once a complete ranking is obtained. The N first individuals are selected for the evolution stage, N being the selection size (parameter of the algorithm).

The use of this approach is simple at it does not require any information from the decision-makers. The result is a set of alternatives well distributed on the first Pareto front. It permits to well understand the problem since exchange rates between criteria and attainable performances are obtained. Nevertheless, because the algorithm aims at preserving the diversity of solutions, the first Pareto front includes solutions that are non-dominated but than may not be interesting for the decision-makers as their preferences are not considered. For instance, decision-makers may look for a compromise alternative with average performances on each objective. Using NSGA-II, extreme solutions (being the best on one objective but bad on others) will be selected whereas the decision-makers' interest zone will comprise only few solutions (because of diversity).

5.2 Preference-based approach

In order to focus the selection of individuals on the inserting zone, a preference-based approach can be used. This kind of approach defines a dominance relation between two alternatives (or individuals) based on their performances and on the preferences of the decision-makers.

We selected the NEMO algorithm [Branke et al., 2009], an interactive preference-based selection algorithm based on ordinal regression. Using this principle, the preferences are deduced from the interpretation of actions (decisions) made by the decision-makers. This mode of preference elicitation requires low cognitive effort from the decision-makers and permit to obtain more pertinent solutions. Indeed, the selected solutions are the closest from the elicited preferences.

The NEMO algorithm is based on the UTA/GRIP methods[Greco et al., 2008; Figueira et al., 2009]. Each evaluation criterion g_i is associated to a marginal value function $u_i(g_i(x))$ which traduces the satisfaction of the decision-makers considering this criterion. In UTA

and GRIP, these value functions are monotone (increasing or decreasing). The value of an alternative is computed as the sum of marginal values.

$$U(x) = \sum_i u_i(g_i(x))$$

During the optimization process, the decision-makers are regularly asked to compare pairs of alternatives and to state [1] :

- the preference of one alternative over the other ($x_1 \succ x_2$)
- the indifference between both alternatives ($x_1 \sim x_2$)

Given this preference information, the marginal value functions can be bounded by putting, among others, the following constraints on the problem:

$$U(x_1) > U(x_2) \text{ if } x_1 \succ x_2$$

$$U(x_1) = U(x_2) \text{ if } x_1 \sim x_2$$

Solving a linear program, a robust partial order (necessary ranking) can be obtained. It sorts individuals in fronts F_i such as, if x_1 is necessarily preferred to x_2 ($x_1 \succ x_2$), then $i < j$. with $x_1 \subset F_i$ and $x_2 \subset F_j$ (the relation $x_1 \succ x_2$ being deduced by the allowed ranges of $U(x_1)$ and $U(x_1)$).

In order to obtain a total order that favor the diversity of solutions, a crowding distance is used. Contrarily to the one defined in [Deb et al., 2000], this crowding distance is not based on the performances of the alternatives but on their most representative marginal values along each criterion.

The use of the NEMO algorithm is interesting as it permits to guide the design-space exploration toward the decision-makers' interest zone. Consequently, the density of solutions present in this zone will be more important than using a Pareto approach and the algorithm will converge faster in terms of solutions value. As it is an interactive method, decision-makers shall be available during the process in order to give preference information.

The selection process permits to select the best architectures of a population with respect to the considered definition of the worth of an alternative. Selected individuals will be used for the creation of a new population during the evolution stage.

1. We only consider the two kind of preference information of UTA whereas additional ones are considered by GRIP

5.3 Considering constraints

In the two above-introduced approaches, only optimization objectives are considered. But optimization problems and, more specially the architecture design problem, is also made of constraints. In our method, some constraints are verified by construction thanks to the use of rules considered during the initialization and the evolution of solutions. Other constraints are considered during selection.

For this, the method introduced in [Deb, 2005] is used. In this method, each constraint has a criticality index which represents its importance relatively to other constraints. The dominance relation is redefined considering the constraint violation (sum of violated constraints criticality) of compared alternatives. Following this definition, optimization objectives are only considered to compare two feasible solutions. In this case, the Pareto-dominance or the robust utility dominance are applied depending on the selected selection process.

CHAPTER 6

TERMINATION AND POST-ANALYSIS

Evolutionary algorithms are iterative processes. Thus, the process requires a criterion to stop this iterative process.

The definition of the stop criterion is very sensitive as it will determine the effort spent to find optimal solutions. If the Stop criterion is too "low", the best solutions found by the algorithm could be actually far from the real optimum as few solutions will have been considered in order to find them. The opposite will lead to explore many solutions (even if the real optimum solutions have been already found) and will increase the computational cost. Generally, the second type of criterion is not used as performances of the architectures are not a priori known and as the process looks for the best solutions and not for the first acceptable solutions found.

The Stop criterion can consider :

- the number of created populations
- the quality of solutions of the current population (e.g. all performances above a threshold)
- the improvement of solutions during a number of populations (e.g. no improvement during N populations)

Once the process is ended, the designers must analyze the results of the optimization. The most interesting solutions i.e. non-dominated solutions are presented to them. With these results, designers are able to :

- check that no evaluable constraints have been omitted
- get the attainable performances of the architectures
- understand the relations between the different objectives
- select the architectures that will be studied in a more detailed trade study.

For this last activity, the designers face a large number of alternatives. In order to decrease the number of architectures to study in details, non-dominated solutions must be ranked based on preferences. If the Pareto selection process was used, preferences must be gathered. If the NEMO algorithm was used, preferences acquired during the optimization process can be reused is this post-analysis ranking process. This must permit to reduce

the amount of information presented for the decision making in order to simplify it. The presentation of this information is crucial and is one of the fields of research directly linked to the here presented works.

In this part, a new method to synthesize and optimize system architectures was presented. This method takes advantage of computer power. Thus, in order to apply it to an engineering problem, a software must be developed. This software is presented in the next part.

Part V

DEVELOPMENTS: THE SAMOA TOOL

CHAPTER 1

INTRODUCTION

In order to demonstrate adequacy of the method with stated needs and its added value to the design process, a tool called SAMOA (System Architecture Model-Based OptimizAtion) was developed. The program was developed in the Java programming language [Java, 2012]. It implements the method (meta-model, synthesis and optimization algorithms,...) and a set of additional tools that are useful to the execution of the design process (visualization, model import and export...). The program is only a demonstrator and do not have the ambition to be industrially usable.

CHAPTER 2

ARCHITECTURE

The tool is constituted of 6 main modules which will be presented in details in the next chapter. Each module can call a number of methods owned by another module as soon as they are declared as public.

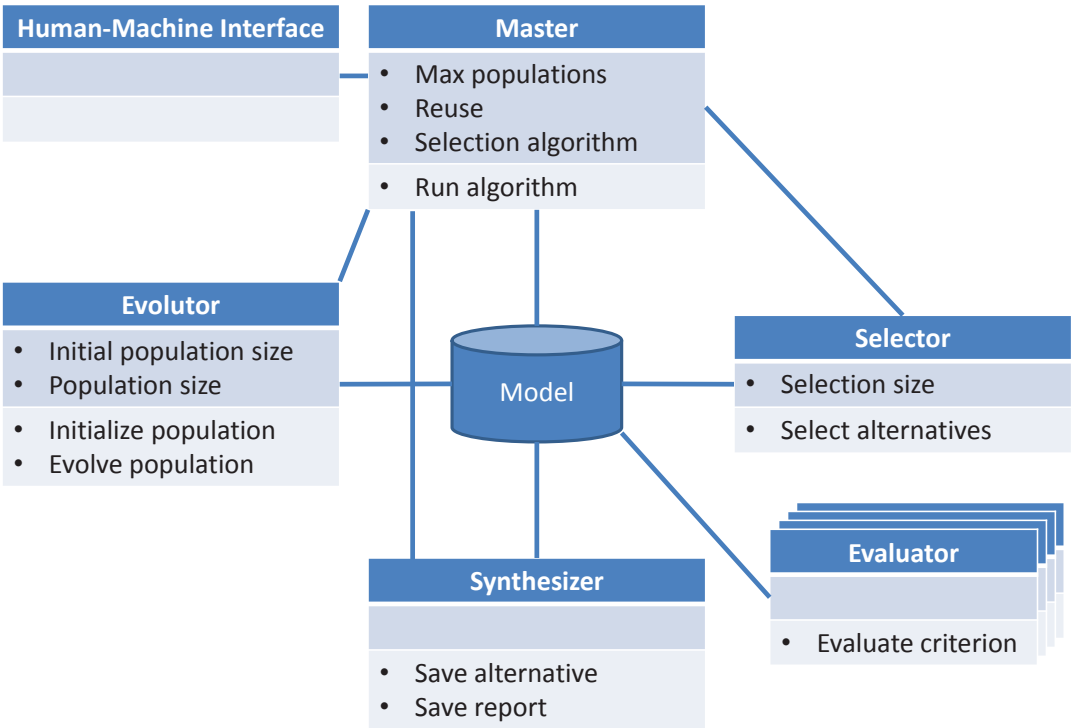


Fig. 2.1: The SAMOA architecture

Data exchange between modules is mainly based on the model which can be read or modified by modules methods. Modules can also exchange data directly through method calls (return).

CHAPTER 3

MODULES

3.1 Master module

The Master module controls the process execution. The *run()* method permits to execute the optimization process and calls sequentially methods of other modules.

The main parameters of the Master module are the *max_pop* and *max_pop_without_improvement* Integer parameters that are used to compute the stop criterion value at each iteration of the algorithm. *max_pop* represents the maximum number of populations and *max_pop_without_improvement* represent the maximum number of population without improvement i.e. without changes in the solution pool (see 3.5). For each parameter, if the parameter value is set to 0, the sub-criterion is ignored.

The Master module is associated to an instance of the 5 following modules.

3.2 Evolution module

The Evolution module contains the methods necessary for the synthesis of architectures (initialization and evolution) among which the chain search algorithm. It uses rules which are methods that return boolean values with respect to their input parameters (chain, sources, sinks, function...).

The Evolution module also contains the methods necessary for the evolution of population during the genetic algorithm loops. The evolution module considers a list of genetic operators derived from two interfaces: single operator and double operator. The first one takes a parent as input and sends back a child where the second one transforms two parent architectures into two children. These interfaces can be seen as templates that can be used to declare genetic operators.

During evolution, genetic operators are selected and applied until the new population reaches its specified size (*pop_size* parameter). The selection of genetic operators is prob-

abilistic i.e. each operator is associated to a selection probability and its chances to be selected is proportional to this probability.

The main parameters of the module are :

- Initial population size : number of alternatives in the first population
- Population size : number of alternatives in other populations

The size of the first population can be larger than the number of alternatives in later populations. This enables to have a large but sparse exploration of the design space at the first iteration and a more limited but guided one in following iterations.

3.3 Evaluation modules

Evaluation modules are specific modules that compute the performance of alternatives along one criteria. For instance, a mass evaluator will gather information in the model necessary to compute the mass of the system, make the computation and give back the computed value. The computation is based on information contained in the model and on computation rules owned by the evaluator.

During the declaration of the model, each criterion must be linked to an evaluation module. For this, the Criterion class owns an abstract method called "evaluate". When the designer creates an instance of the Criterion class, the method must be overridden by declaring how the evaluation of the criterion must be performed. For readability of the model, this method is often a simple call to another method. When a method wants to know the performance of an alternative on a given criterion and that the performance has not yet been computed, the evaluate method is called and returns the performance. Thus, evaluations are performed on-demand, which permits to avoid useless computations.

In order to be flexible, the task of developing evaluators or of interfacing existing tools to evaluate the performances of alternatives is left to the user. This way, evaluators can be adapted to the context. For instance, the level of precision of the calculation can be tuned and can consider different types of data contained in the models. The mass evaluator can be a simple sum of component masses but can also be computed taking into account location of components (for cable mass computation).

3.4 Selection module

The Selection module is responsible for selecting the "best" individuals of a population. The two algorithms presented in 5 (NSGA-II and NEMO) are contained in the module and can be used by the Master module. In case NEMO is selected, preference information is stored in the model (Preference and Indifference objects).

The main parameter of this module, additionally to algorithm to be used, is the size of the selection i.e. the number of individuals to select. This parameter must naturally be lower to the size of the population.

3.5 Solution pool

In order to keep track of "best" alternatives, a solution pool is set up. The pool is used to store the Pareto solutions of the problem considering all solutions discovered so far (not only the last population).

When a new population is created, the pool is updated. For this, each individual of the population is compared to the solutions of the pool. If a new solution dominates one or several pool solution(s), the former replaces dominated solution(s) in the solution pool. The dominance definition used in the solution pool is the Pareto one.

The solution pool also permits to determine the number of populations without improvement. Indeed, when the pool is updated with a new population and is modified by this update (i.e. new solutions are inserted), a counter is reseted. Instead, it is incremented. This counter is used by the Master module to determine the value of the stop criterion at each iteration.

3.6 Human-Machine Interface

In order to facilitate the method use by system designers we developed a Human Machine Interface (HMI). This HMI uses the JavaFX 2.0 platform and provides a visual support for:

- the analysis of the problem model in order to check its correctness,
- the parameterization of the algorithm through an access to modules parameters,
- the process monitoring,
- the analysis of results.

It is organized in three modes : model, process, results.

3.6.1 The Model mode

The Model mode permits to import models and to visualize them.

A Model defined using a Java method returning a Model instance can be directly imported. Models can also be defined in other formats. In this case, a parser must be used to translate the elements of the model in the format into Java objects (see 3.7.1).

The content of the imported model can be visualized as tables displaying elements of the problem (functions, components...) and their properties.

3.6.2 The Process mode

The process mode has two functions : setting the parameters of the algorithm and monitoring the optimization process.

Parameters of the process that can be modified in this mode are:

- Initial population size
- Population size
- Selection size
- Selection algorithm (NSGA-II or NEMO)

The monitoring of the optimization process is done by plotting the evolution of maximal, minimal and mean performances at each iteration and for each criterion. This permits to visualize the convergence of the algorithm on every criteria.

Additionally, the user can visualize the performances of the last population as a scatter-plot matrix. The scatter-plot matrix A contains all the pairwise scatter plots of the variables i.e. each cell a_ij of the matrix contains a scatter plot of the performances of the population on the criteria i versus the performances of the population on the criteria j. The diagonal of the matrix (a_ii) contains histograms of the performances of the population on the criteria i.

3.6.3 The Result mode

The Result mode permits to analyze individually solutions. For this, a table presents the performances of all alternatives of a population. The user can navigate through the populations (including the solution pool) and can access to an architecture. Then, the user can visualize the selected architecture as a graph of component instances and system ports (physical view) and can access the information of the components (function allocations, variables, ports, attributes...), of the system ports and of connections.

In the Result mode, the user can export current results:

- as an Excel sheet for alternatives performances
- as images for the architecture physical view
- as text file for the architecture complete view
- as matrices (DSM+FCM) for the architecture complete view

3.7 Other tools

3.7.1 Parsers

In order to be able to import models described in various formats, parsers must be used to translate them into Java Model instances. Parsers read an input file, locate objects and their class and create a Java instance of the same class.

In the current development state, the user can import models defined by:
- a Java method returning an instance of the Model class (does not require a parser)
- an Excel sheet built from an Excel template
- an XML file

In the same way, Parsers can be used to export the current Model (e.g. the model after optimization with all populations) or part of it. For instance, different Parsers are used to export results in the Result mode (see 3.6.3).

3.7.2 SysML interface

SysML is becoming the standard modeling language for systems and the cornerstone of MBSE approaches. In this context, it is interesting to be able to model the problem with this language and to import it into the SAMOA tool. To do so, a capability was developed.

SysML modeling

In order to model the problem is SysML, the IBM Rational Rhapsody platform is used. However, since many of the meta-model concepts do not have equivalents in SysML, a profile needed to be defined. A profile permits to define new concepts based on existing SysML concepts.

Using the SAMOA profile, a model of the problem can be built by declaring the necessary elements (functions, component types...). One of the main advantage of SysML is that the problem can be represented using several views (e.g. optimization view, system view, component types view). A user guide for the SAMOA profile can be found in Annex C.

SysML model import

The import of the model i.e. the transformation of the SysML objects into Java instances is made thanks through the use of the Rhapsody Java API. The API permits to browse the SysML model and, for each element of the model, to create an equivalent Java instance.

The API would also permit to enrich the SysML model with architecture solutions found by the SAMOA tool. This capability was not developed.

CHAPTER 4

MODELING

The meta-model presented in 2.3 was implemented in the Java programming language. Thanks to its object-oriented property, this language permits to create instances from meta-classes.

Each meta-class is a Java class and contains:

- properties of the meta-class (e.g. name)
- relations to other meta-classes
- methods

The following presents an example of implementation of a class of the meta-model : the Architecture class. The class has 6 fields (ID,description, functions, ports, components and connections), 2 constructors (a simple one and a one that clones an architecture given as parameter) and 48 methods (among which the *getAllocatedComponents()* one which gathers components allocated to a given function).

```
package MetaModel;

import static com.wagnerandade.coollection.Coollection.*;
import com.wagnerandade.coollection.Coollection;
import java.util.*;
import Modules.Check;
import Modules.Rules;

public class Architecture implements Cloneable{

  Integer ID;
  List<Function> functions= new ArrayList<Function>();
  List<Port> ports = new ArrayList<Port>();
  List<Component> components = new ArrayList<Component>();
  List<Connection> connections = new ArrayList<Connection>();
  String description;

  public Architecture() {
```

```
19      super();
20    }
21
22    public Architecture(Architecture archi) {
23    [...]
24    }
25
26    [...]
27
28    public HashSet<Component> getAllocatedComponents(Function function) {
29      HashSet<Component> comps = new HashSet<Component>();
30      for (Component c : this.components){
31        if (c.getAllocationsByName(function.getName()) != null)comps.add(c);
32      }
33      return comps;
34    }
35
36    }
```

Listing 4.1: Implementation of the Architecture class in the Java programming language

Methods are used to read and aggregate information contained in classes or to modify them. In Java, getters and setters respectively permit to read and modify a data contained into a class.

Based on the set of Java classes, models of design problems can be built. For this, the user must create instances of meta-classes and fill in instance properties using constructor's parameters.

```
1 archi.getComponents().add(new Component("sDDU_1",model.
    getComponentTypeByName("sDDU")));
```

Listing 4.2: Creation of a component in an Architecture *archi*

Part VI

APPLICATION

CHAPTER 1

PRESENTATION OF THE COCKPIT USE-CASE

1.1 Introduction

The cockpit system is the system permitting the control of the main functions of the aircraft (flight control, main systems control, . . .) and the monitoring of the aircraft state by pilots. It is generally located at the front of the aircraft and provides control means (control panels, levers, touchscreens,. . .) and visualization means (screens, lights, . . .).

Fig. 1.1: A380 cockpit

The cockpit system is exchanging data with the different systems of the aircraft in order to monitor and control their state. For instance, in case of a fire detection, the Fire Detection System sends an alarm message to the cockpit system. The cockpit system displays this alarm to pilots which will react in consequence following the procedure and will activate the Fire Extinction System in the impacted zone of the aircraft by acting on the "Fire" control mean (button, virtual button on a touchscreen or even vocal command). The Fire Extinction System will react to this order by activating its fire-fight means.

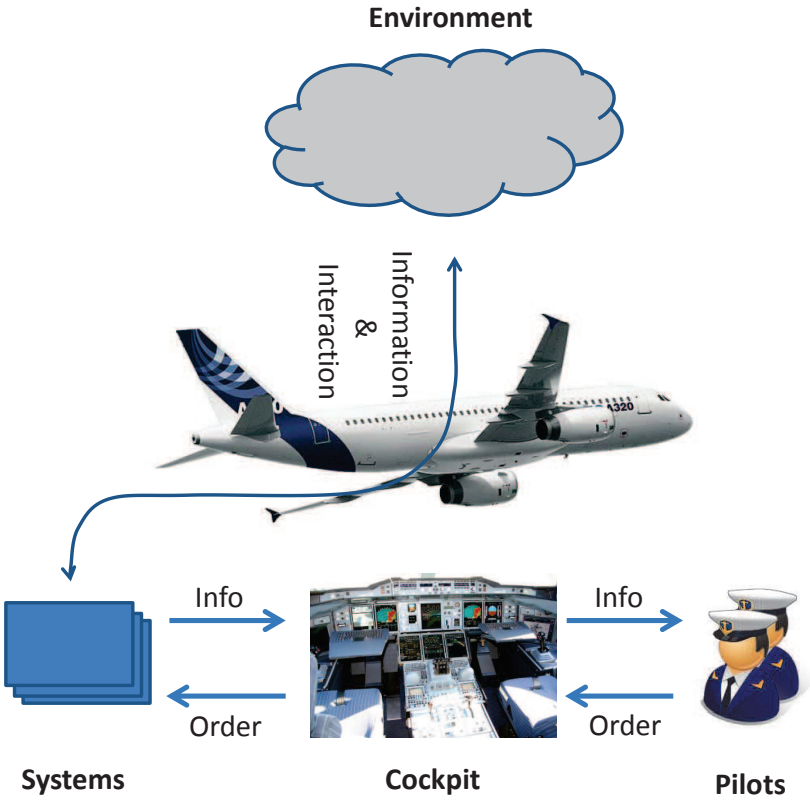


Fig. 1.2: Cockpit interfaces

1.2 Functions

The main functions of the cockpit are:

- To inform the pilot about the past, current and future state of the aircraft. The information includes displays (information available all along the flight) and alerts (punctual information given only in abnormal situations). Displayed information includes attitude, altitude, speed, position, flight plan, state of systems...
- To gather pilots orders and send them to aircraft systems (control)

These functions are critical as their failure (loss or erroneous behavior) can have hazardous to catastrophic consequences (as defined by the Federal Aviation Administration [FAA, 2000]) . This induces that the realization of the cockpit functions are subject to numerous safety constraints. For instance, the failure of a single component in the architecture must not lead to the loss of functions with a catastrophic severity. Safety constraints have major impacts on the design of the system architecture as they impose the existence of redundancies.

The perimeter of this study only considers the interface functions of the cockpit (display, control, alert) and does not include other functions such as functions related to pilot comfort (cockpit lightening, seats...), to pilot vision, and to communications with cabin crew or Air Traffic Controllers (except the *ATC_Mail* function).

1.3 Realizations

The functions of aircraft cockpits are quite fixed and stable in time. Only the representation of information varies from one manufacturer to the other. Also, the possibilities to realize these functions are quite weak: the display of information must be done through a screen or an instrument, the control of the on/off state of a system must be done through a button. For this reason, one can note an apparent similarity between the cockpits of different aircraft since the 80s. Some evolutions can be highlighted:

1. One can note the tendency to virtualize instruments (i.e. to replace dedicated instruments by a displayed equivalent). Today, few critical instruments remain in current cockpits for safety reasons (redundancy and dissimilarity) and are used as backup information source. On A330, 4 analog instruments were present whereas on A380, no mechanical or analogical instrument is present. The Integrated Standby Instrument System (ISIS), that provides standby information to pilots, is made of two screens that are able to display different kind of simple information.
2. The number of information to be displayed and the complexity of their representation is growing with the complexity of the aircrafts. Thus, the display surface is

significantly growing and the display technologies are evolving (monochrome to color display units).

3. New equipment appeared to increase or simplify interactions with pilots (keyboards, KCCUs. . .)

If the visible part (HMI) of the cockpit system seems quite stable in time, the general architecture of cockpits is also evolved very slowly until the 2000s. At this time, new architectures were designed based on the Integrated Modular Avionics (REF) concept. This concept aims at computational resources in order to

Nowadays, as needs evolve and large technological advances are done, cockpit designers must consider new concepts involving new technologies and new interaction modes with pilots. . . . An example of an innovative concepts is the ODICIS single-display interactive cockpit [2] which takes advantage of touch-screen capabilities.

1.4 Design process

The design of a cockpit system is a complex task because:

- the system itself is complex (numerous functions)
- the system requirements are constraining notably in terms of reliability, availability, maintainability and safety (RAMS)
- the system is an HMI and human factors must be considered with care.

The component types available to the designers are few. Nevertheless, the possibilities to assemble these components and to allocate functions to them (i.e. the design-space) are huge.

A current solution to create design alternatives is to take previous or existing architectures and to modify them to handle new functions, to integrate new technologies. Doing so, new architectures are created but, as they are all issued from the same parent, they are quite similar.

Another solution is to build a decision tree in order to obtain a lot of potential solutions. Then, branches are analyzed by the designers on the base of their knowledge to simplify the tree until it becomes humanly understandable. A try was given on the next Airbus program. A filtered version of the tree gave more than 2000 alternatives only considering physical aspects (no allocations of functions). Nevertheless, as the elaboration of the tree was only based on designers´ knowledge and believes, many alternatives were not considered (notably radical innovative ones). For instance, the single-display architecture was not considered. Once alternatives have been identified, designers try to evaluate their

capability to fulfill system requirements and compare their performances. Different criteria are taken into account:

- Costs (recurrent and non-recurrent)
- Weight (direct and induced)
- Energy consumption
- Operational performances (latencies...)
- Dependability (reliability, availability, maintainability, safety, integrity)
- Robustness to environmental conditions (heat, dust...)
- Complexity
- Evolution capabilities
- ...

These criteria are technical criteria that can be assessed quite easily even on an ordinal scale. Other criteria are more difficult to handle as they deal with economical, industrial or societal aspects.

A criterion is particularly important for assessment of HMI but quite difficult to assess: the "human factor" criterion. This criterion deals with the ease of usage of the system. Its assessment is quite difficult as it must take into account the human whom behaviour is very complex. The assessment of the system performance with respect to the human factor criterion is mainly based on expertise of ergonomics experts. Some attemps to build models of the human-machine interaction were made in the last few years. This models try to estimate the cognitive load of pilots with respect to a given HMI configuration, scenarios and environement conditions.

A big difficulty of system architecture design, and specially for cockpits considering the technological progresses in HMI systems (touchscreens, mobile systems, wireless systems...), is the technological foresse Indeed, in the aeronautical field, architecture design begins several years before the first flight. During the design process, technologies will appear and customer expectations may evolve. Designer must consider this by anticipating the emergence of new technologies and of new customer needs.

CHAPTER 2

MODEL

2.1 Model presentation

In this Chapter, the method will be applied to the cockpit use-case. For this, a model of the problem was built. Even if it does not contains all elements of the real problem (functions, constraints...), it is representative of the complexity of the problem.

2.1.1 System

The system model consists in 9 ports which represent the interfaces of the system with its environment :

- Pilot interfaces
  - 2 visual interfaces (IC and IF) representing the eyes of each pilot through which flows of the $Image$ type can be received (output port).
  - 2 haptic interfaces (CC and CF) representing the hands of each pilot through which flows of the $Control$ type can be sent (input port).
- Systems interfaces (for readability of the model and of the results, a single interface of each type for all systems is modeled)
  - 1 system input interface (Sys_out) through which flows of the $Data$ type can be received by systems.
  - 1 system output interface (Sys_in) through which flows of the $Data$ type can be sent by systems.

2.1.2 Functions

The model, in its most complex version, is constituted of 19 independent functions (i.e. not exchanging flows).

Display functions

10 display functions are described in the model. These functions take raw data from aircraft systems as input, generate an image from them and display this image to the pilots (Captain and Flight Officer F/O). We will consider that the image must be displayed to at least one pilot so there is only one image output to these functions. Display functions have up to 35 input data flows. As display functions perform data aggregation and graphical generation, they request a *computing* capability.

As an example, we present a display function Navigation Display (ND) which displays the information relative to the mission on the horizontal plan (flight plan, Instrument Landing System (ILS), VHF Omni-directional Range (VOR),...) .

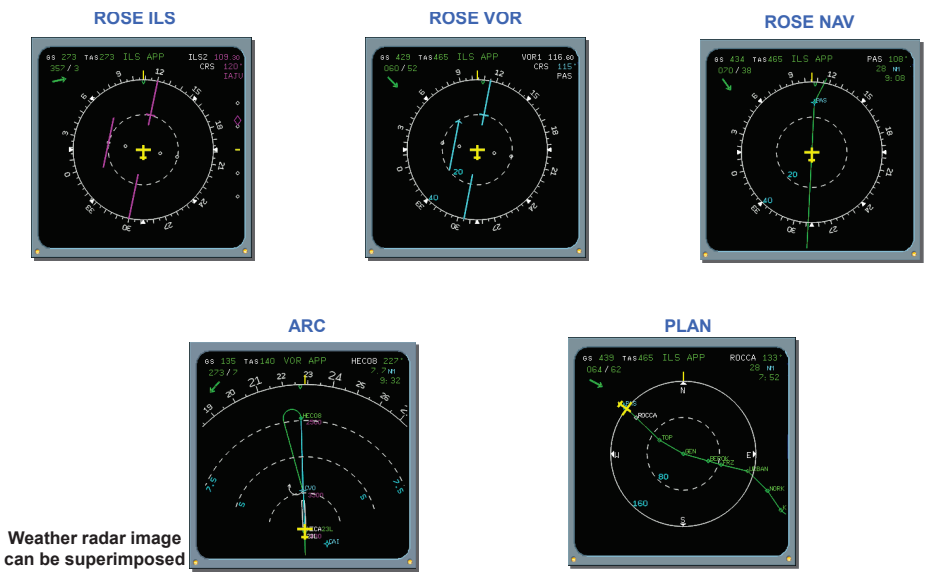


Fig. 2.1: Navigation Display

The function receives data from 12 systems : Automatic Direction Finder (ADF) , Air Data and Inertial Reference System (ADIRS), Aircraft Environment Surveillance System (AESS), Distance Measuring Equipment (DME), Electronic Flight Instrument System (EFIS) Control panel, Flight Control and Guidance Unit (FCGU), Flight Control Unit Back-Up (FCU-BU), Flight Management System (FMS), Flight Warning System (FWS), Multi Mode Receiver (MMR), Onboard Airport Navigation System (OANS) and VOR. These data are aggregated and, depending on the current mode, the information is displayed (ND_i_Capt and ND_i_FO of type $Image_Capt$ and $Image_FO$).

Function acronym	**Function name**	**Description**
ATC_Mail	Air Traffic Control Mailing Box	Displays messages form Air Traffic Control
Engine	Engine status and performances	Displays engine status (power rate, oil temperature...
MFD	Multi-Function Display	Displays miscellaneous information (Aircraft Environment Surveillance, Air Traffic Control...)
ND	Navigation Display	Displays the horizontal mission profile and related data (traffic, weather...)
OIS_Video	Onboard Information System Video	Displays videos from the open world (e.g. cabin cameras)
PFD	Primary Flight Display	Displays speed, altitude and artificial horizon
SD	System Display	Displays the state of the aircraft systems
SFTD	Slats, Flaps and Trim Display	Displays the state of slats, flaps and trim
VD	Vertical Display	Displays the vertical mission profile
WD	Warning Display	Displays warnings

Tab. 2.1: List of Display functions

Control functions

9 control functions are described in the model. These functions take inputs from the pilots (button push, button rotation, lever action...) and transmit them to aircraft systems. Controls can be given by the two pilots (captain or F/O). Consequently, control functions have two control inputs, one for each pilot. An example of control function is depicted below : the Anti-ice control function which permits to control the anti-icing system.

Fig. 2.2: Anti-Ice control function

Function name	Description
Comfort	Control comfort parameters (loudspeakers' volume, screens' brightness...)
Braking	Control braking modes
Engine	Start/Stop engines and Auxiliary Power Unit
Fuel	Select the fuel source
Fire	Activates fire fight means
Smoke	Activate smoke evacuation means
Air	Select the air sources
Power	Select the power sources
Anti-ice	Control anti-ice systems

Tab. 2.2: List of Control functions

2.1.3 Component types

We consider 18 component types among which 5 are abstract. A detailled description of component types can be found in Appendix A.

Each HUD has a *Control* induced function that represents the need of additional components to control its functioning (brightness, alignment, etc.). During the availability analysis, the hypothesis is made that the failure of any induced function of a component leads to its failure.

	Component type	Description	Abstract
Displays	DU	Display Unit	Yes
	sDU	Shared DU	Yes
	pDU	Private DU	Yes
	sDDU	Shared Dumb DU	
	sSDU	Shared Smart DU	
	pDDU	Private Dumb DU	
	pSDU	Private Smart DU	
	HUD	Head-Up Display	
	CPU	Central Processing Unit	
	GPU	Graphical Processing Unit	
	ICP	Integrated Control Panel	
Cables & Networks	Cable	Generic cable	Yes
	dC	Data Cable	
	eC	Electrical Cable	
	NW	Network	Yes
	CAN	Controller Area Network	
	ADCN	Avionics Data Communication Network	
	eNW	Electrical Network	

Tab. 2.3: List of component types

Control panels (ICP) are generally custom parts dedicated to the function(s) they realize. For instance, the over-head panel contains buttons for the different functions it realizes (fire, smoke, air, etc...). Its attributes (e.g. mass) depend on its allocated functions. In the model, as a simplification, we only consider a generic ICP with fixed attributes whatever the functions it realizes.

2.1.4 Criteria, objectives and constraints

In the frame of this study, we consider 5 quantifiable criteria that permit to determine the relative quality of an alternative. Each criterion is associated to an evaluator which computes, for each alternative, its performance along the associated criterion.

Mass

The *Mass* criterion represents the direct mass of the system (the induced mass is not considered). The evaluator sums the mass of each component of the system.

For cables and networks, the mass is computed based on the length and on the linear mass of the cable/network. The length is computed by the evaluator based on the location of the connected components and on a table giving an average cable length between two locations. In the model, the *Location* attribute takes two values (cockpit or bay).

For other components (DU, ICP, etc.), a *Mass* attribute value is considered.

This criterion is a classical one in the aeronautical industry as increasing the mass increases recurring costs and has major impacts on the overall aircraft design (airframe and engines sizing). We associate this criterion to a minimization objective.

Component number

This criterion is the sum of components present in the architecture.

The number of components must be minimized to decrease the complexity of the system and the integration costs (non-recurring costs).

Part number

This criterion is the sum of component types used in the architecture. For instance, if an architecture contains 2 DUs and 1 CAN, its part number is 2 whereas its component number is 3.

The part number must be minimized to decrease the complexity of the system and the non-recurring costs (integration, acquisition, maintenance).

Electrical consumption

This criterion is the sum of components mean electrical consumption. Components (except cables and networks) have a *Consumption* attribute.

This criterion is associated to a minimization objective as it permits to decrease the fuel consumption and to reduce the sizing of the power system.

Minimum availability

This criterion represents the minimum availability of output functions computed as their Mean Time To Failure (MTTF) in hours.

To evaluate the MTTF of a function, the evaluator proceeds in the following manner :

Algorithm 7 Function MTTF computation algorithm

```
Search paths realizing the function F and its supplier functions
Determine minimal cut-sets mC_i(F)
for all Components C do
  if Component has an induced function then
    Search paths realizing the induced function IF and its supplier functions
    Determine minimal cut-sets mC_i(IF)
    for all Previous minimal cut-sets mC_i(F) do
      Clone the cut-set and replace C by mC_i(IF)
    end for
  end if
end for
Compute MTTF of the function MTTF(F)
```

The MTTF of the function is computed based on the MTTF of the components ($MTTF_c$) present in the function minimal cut sets ($mC_i(F)$) as follows :

$$MTTF(F) = \frac{1}{\sum\limits_{mC \in mC_i(F)} \prod\limits_{c \in mC} \lambda_c}$$

where

$$\lambda_c = \frac{1}{MTTF_c}.$$

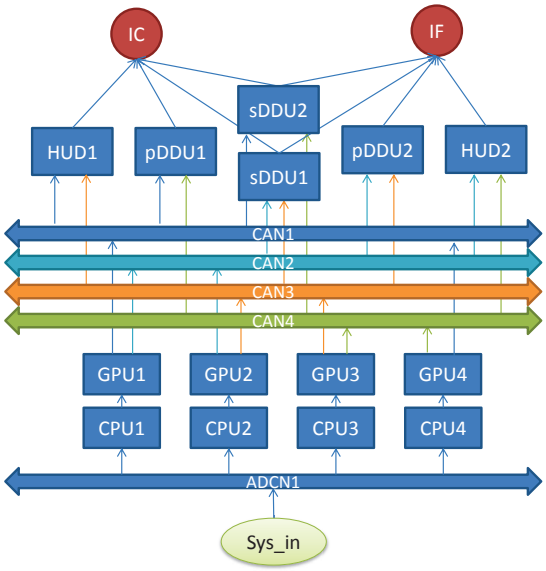
2.2 Reference architecture

In order to give a reference point and to be able to compare the results of the method to the results of a classical development process, a reference architecture is presented here after. This architecture is inspired from a real architecture considered in the development process of an Airbus aircraft. Only the functions considered in the model and their allocated components are kept.

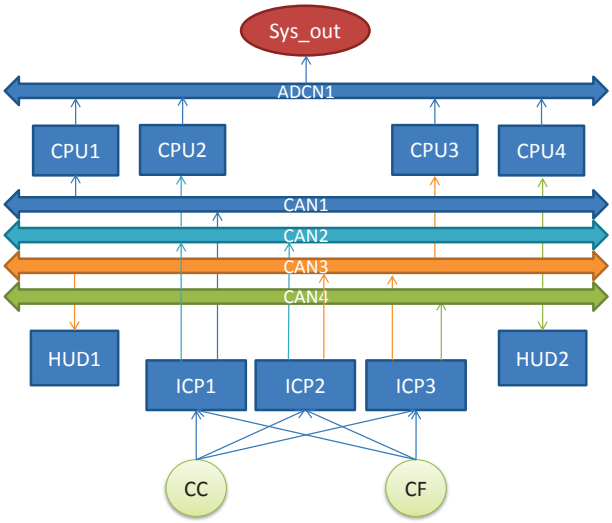
The physical architecture is depicted in Figure 2.3.

Display functions are allocated to 2 CAN, to 2 GPUs and to 2 CPUs to increase their availability. The availability of a display function is very close from the MTTF of the display unit allocated to it (64900h for a DDU, 38000h for a SDU or for a HUD).

Performances of the reference architecture are presented in Table 2.4. The availability analysis is presented in B. This analysis only considers the nominal behavior of the cockpit system i.e. the allocation of functions when no component is failed. Of course, in the real



(a) Display functions



(b) Control functions

Fig. 2.3: Reference architecture

system, reconfigurations of the system occur in case of failure and permit to increase the availability of critical functions and to satisfy safety requirements.

Of course, the reference architecture must not be considered as an optimal one for the problem considered here as:

- it was designed to satisfy the actual system requirements that are not consider exhaustively in our model,
- it was designed considering other constraints (industrial, human-factor, economical...) that are not considered in our model.

It must be more understood as a reference point to compare the synthesized alternatives to an alternative issued from a human-based design process.

Criteria	Performance
Component number	22
Part number	8
Electrical consumption (W)	873
Mass (kg)	57.3
Min availability of functions (h)	38000

Tab. 2.4: Performances of the reference architecture

2.3 Results

2.3.1 Population initialization

Without reuse mechanism

A population of 1000 alternatives is created thanks to the initialization algorithm. The reuse probability is set to 0.0 so that the algorithm considers component types and components the same way during the search of chains. The process took 1h40 on a standard laptop [1]. This duration comprises synthesis and evaluation of alternatives but also IHM update and report generation.

Among the 1000 synthesized architectures, 118 do not violate any constraint. On these remaining architectures, 6 are Pareto solutions.

The results of this run show that the algorithm is able to generate solutions with various performances except for the *Min availability* criterion. Indeed, during initialization, no redundancies are created and the solutions to realize control functions have a low availability time.

1. Intel Core 2 Duo CPU P8600 @2.4GHz with 2Go RAM

	System mass	Part number	Components number	Mean electrical consumption	Min Availability Time
Ind 100	57	8	32	959	27055,7029
Ind 105	53,7	9	27	857	27055,7029
Ind 154	52,1	9	26	739	27055,7029
Ind 203	60,9	10	40	1011	27481,0491
Ind 693	54	8	33	873	26760,0737
Ind 752	69	7	33	1021	26760,0737
Ref	57.3	8	22	873	38000

Tab. 2.5: Pareto solutions after population initialization (reuse probability = 0.0)

Comparing the results to the reference architecture performances, one can notice that the ranges of performances obtained by synthesis include the performances of the reference architecture except on the *Min availability* criterion (for the aforementioned reason) and on the *Component number* criterion. For this last criterion, this is due to the fact that the algorithm does not seek to reduce the complexity of architectures by reusing existing component instances. Indeed, during the search of chains, components and component types are considered in a completely random order. In order to force the algorithm to synthesize less complex architectures, the *Reuse probability* can be increased.

With reuse mechanism

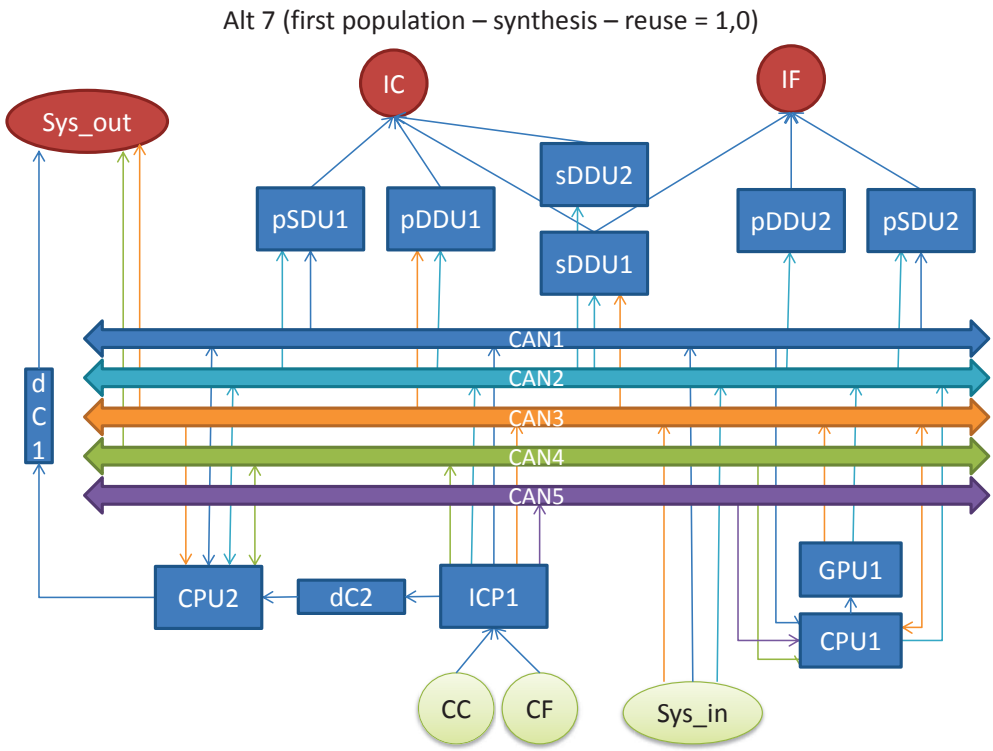
A population of 1000 alternatives is created thanks to the initialization algorithm. The reuse probability is set to 1.0 (existing components are considered before new instances of component types). The process took 46 min.. On the 1000 generated alternatives, 236 do not violate any constraint. Among these acceptable solutions, 14 are non dominated (in the Pareto sense). The results are presented in Table 2.6 and a Pareto solution is depicted in Figure 2.4.

Comparing the obtained solutions to the reference architecture, one can notice that the performances of the synthesized architectures are equal to or better than the reference one on the four first criteria. Only the minimum availability of the reference architecture is better than the generated solutions. This is due to the fact that no redundancies are created by the synthesis algorithm.

The complexity of synthesized architectures is decreased thanks to the increase of the *Reuse probability* parameter. Concerning performances of the algorithm, an outcome is that the *Reuse probability* mechanism permits to reduce the computation time. Indeed,

	System mass	**Part number**	**Components number**	**Mean electrical consumption**	**Min Availability Time**
Ind 7	40,4	8	17	576	27055,7029
Ind 165	41,7	9	18	619	27792,9155
Ind 306	37,4	9	16	626	27481,0491
Ind 526	39,9	8	20	728	27055,7029
Ind 530	37,3	9	22	642	27055,7029
Ind 557	37,4	7	23	626	27481,0491
Ind 616	43	8	14	662	27792,9155
Ind 617	44	7	18	812	27055,7029
Ind 648	38,6	7	22	685	27055,7029
Ind 654	42,5	8	19	678	27481,0491
Ind 706	43	8	22	662	27055,7029
Ind 725	33,4	7	24	649	27481,0491
Ind 785	42,5	7	19	678	27055,7029
Ind 930	44,5	7	16	796	27481,0491
Ref	57.3	8	22	873	38000

Tab. 2.6: Pareto solutions after population initialization (reuse probability = 1.0)



(a) Physical architecture

DMM Alt7	CAN_1	CPU_1	GPU_1	CAN_2	pDDU_1	CAN_3	CPU_2	pDDU_2	pSDU_1	sDDU_1	sDDU_2	pSDU_2	ICP_1	dC_1	dC_2	CAN_4	CAN_5
ATC_Mail	x	x	x	x	x	x	x	x									
Engine			x	x		x	x		x	x							
MFD	x	x	x	x		x					x	x					
ND				x					x			x					
OIS_video		x	x	x		x	x	x		x							
PFD	x		x	x	x	x	x	x									
SD	x			x					x			x					
SFTD	x	x	x			x		x	x								
VD	x		x	x		x	x		x	x							
WD	x			x					x			x					
C_comfort						x	x						x	x			
C_braking	x	x				x							x				
C_engine				x			x						x		x		
C_fuel		x				x							x			x	
C_fire		x		x									x			x	
C_smoke				x			x						x			x	
C_air	x						x						x			x	
C_power				x		x	x						x				
C_antiice		x		x									x				x

(b) Allocation table (DMM)

Fig. 2.4: A Pareto solution after initialization (reuse probability = 1.0)

as less components are present in the architecture, the combinatorics for possible chains is reduced.

When looking at the solution presented in 2.4 in details, one can notice some undesirable structures of components. For instance, $CAN5$ is connected as a cable (one input connection and one output connection). From an engineering perspective, this is incorrect as a network must link more than two equipments. However, as networks have generally a higher mass and a lower MTTF than cables, during optimization, they might be replaced by cables or they might be merged with other networks with more than two connected components.

2.3.2 Optimization

Mutation only

The first run of the optimization algorithm permits to see the effect of the optimization using only the mutation mechanism.

Initial population size	500
Population size	100
Max. population	10
Selection size	20
Reuse	1.0
Operators	Mutation

Tab. 2.7: Parameters of the first run (mutation only)

By comparing these results (Table 2.8) to the previous ones, one can note that the solutions obtained by optimization (except the initial alternative 89) are not dominated by solutions presented in Table 2.6. This observation confirms that the optimization mechanism permits to discover better architectures.

Mutation and redundancy creation

A second run of the algorithm will show the effect of redundancy creation on the Availability criterion.

On the 2500 synthesized architectures, 127 are Pareto solutions. For clarity, we removed from Table 2.10 architectures with equal performances.

The results of this run show that the redundancy creation mechanism as a low effect on the *Min availability* criterion. This is due to the fact that, in order to increase significantly

	System mass	Part number	Components number	Mean electrical consumption	Min Availability Time
Ind 89	48,1	8	14	714	27792,9155
Ind 502	42,6	9	17	662	28571,4278
Ind 541	36,6	10	14	567	27792,9155
Ind 580	31,5	8	15	515	27792,9155
Ind 609	42,6	9	17	662	28571,4278
Ind 669	36,6	10	14	567	27792,9155
Ind 671	30,2	9	13	472	27055,7029
Ind 646	31,5	8	15	515	27792,9155
Ind 686	31,5	8	15	515	27792,9155
Ind 693	31,5	8	15	515	27792,9155
Ind 698	31,5	8	15	515	27792,9155
Ind 704	36,6	10	14	567	27792,9155
Ind 725	31,5	8	15	515	27792,9155
Ind 776	31,5	8	15	515	27792,9155
Ind 790	31,5	8	15	515	27792,9155
Ind 819	31,5	8	15	515	27792,9155
Ind 924	31,5	8	15	515	27792,9155
Ind 1145	36,1	6	15	583	27792,9155
Ind 1187	36,1	6	15	583	27792,9155
Ind 1259	36,1	6	15	583	27792,9155
Ref	57.3	8	22	873	38000

Tab. 2.8: Pareto solutions after optimization (Mutation)

Initial population size	500
Population size	100
Max. population	20
Selection size	20
Reuse	1.0
Operators	Mutation + Redundancy creation (same probability to be applied)

Tab. 2.9: Parameters of the second run (mutation + redundancy creation)

	System mass	Part number	Components number	Mean electrical consumption	Min Availability Time
Ind 24	42,5	7	15	678	27792,9155
Ind 235	49	8	15	757	28571,4278
Ind 349	35,9	8	18	751	27055,7029
Ind 366	37,9	8	15	610	27055,7029
Ind 421	36,2	10	22	567	27792,9155
Ind 537	41,9	7	20	710	27055,7029
Ind 829	37,8	9	15	626	27792,9155
Ind 871	33,5	9	19	633	27055,7029
Ind 886	32,9	8	24	665	27792,9155
Ind 846	33,5	9	20	633	27792,9155
Ind 1537	36,1	8	16	583	28571,4278
Ind 1777	40,4	8	16	576	28571,4278
Ref	57.3	8	22	873	38000

Tab. 2.10: Pareto solutions after optimization (Mutation + Redundancy creation)

this performance, all control functions must have redundant chains, which would require the redundancy creation mechanism to be applied on one architecture as many times as there are control functions in the model. Obviously, this probability is quite low.

Nevertheless, at a local level (i.e. for a function), the application of the redundancy creation operator permits to increase the availability of functions. For instance, it can increase the availability of a control function from 27792h to 40000h (see Figure 2.5).

2.3.3 Introduction of crossover

In this run, the crossover mechanism is introduced. The results (Table 2.11) show that the crossover acts as a catalyst for the optimization process. Indeed, whereas single-parent genetic operators explore the design-space locally, crossover permits to jump between different areas of the design-space.

The architecture 2067 is presented in Figure 2.6. One can notice that the functions of the system are realized by few components. This can be explained by two reasons :

1. As the algorithm do not manage to increase the *Min Availability Time* criterion, the creation of redundancies penalizes the generated architectures on other criteria. Consequently, the algorithm will tend to keep architectures without redundancies.

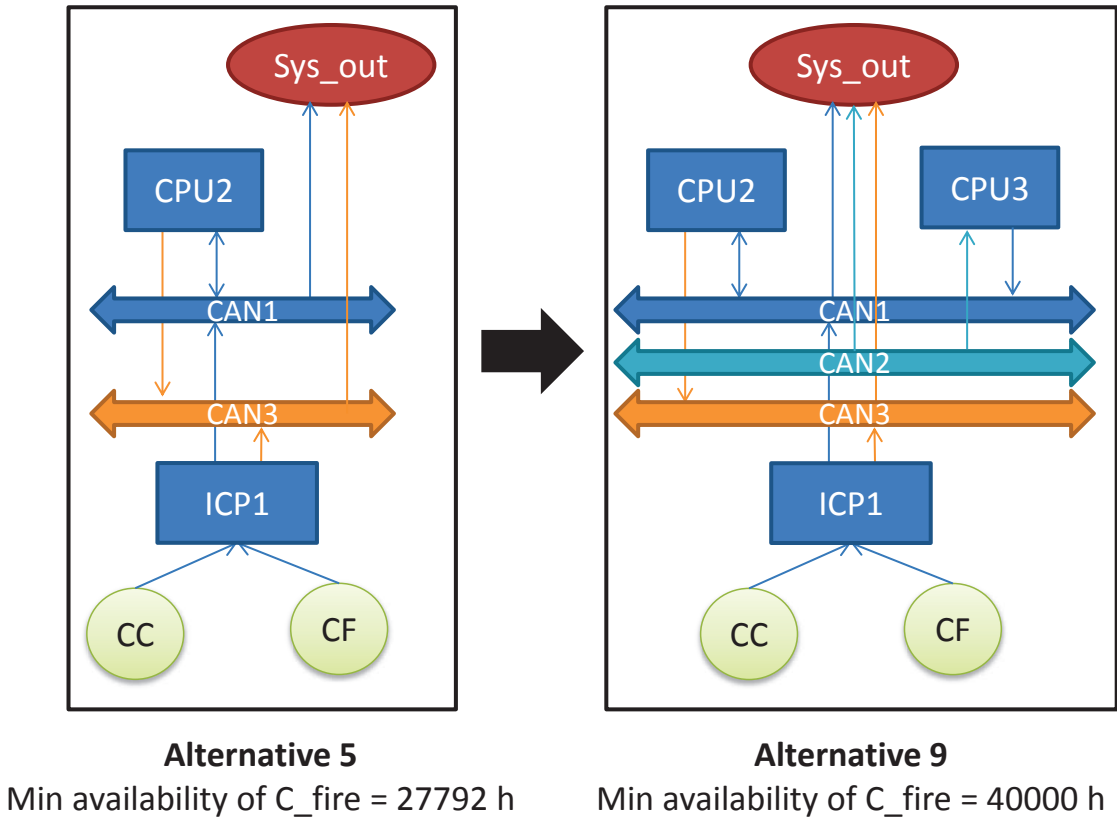
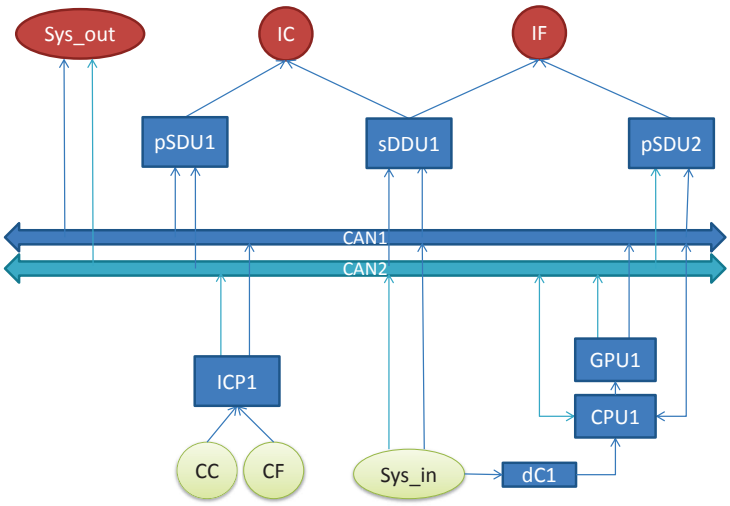


Fig. 2.5: Effect of the redundancy creation operator on the availability of the C_fire function

2. No constraint is set on the segregation of functions so, ideally, only two chains of components are needed : one for display functions and one for control functions.

	System mass	Part number	Components number	Mean electrical consumption	Min Availability Time
Ind 166	41,8	9	16	603	28571,4278
Ind 844	36,2	7	16	567	28241,9495
Ind 2067	23,9	6	8	361	28241,9488
Ref	57.3	8	22	873	38000

Tab. 2.11: Pareto solutions after optimization (Crossover + Mutation + Redundancy creation)



(a) Physical architecture

Alt2067 (DMM)	sDDU_1	CAN_1	GPU_1	CPU_1	dC_1	CAN_2	pSDU_1	pSDU_2	ICP_1
ATC_Mail	x	x	x	x	x	x			
Engine	x	x	x	x		x	x		
MFD	x	x	x	x		x	x		
ND						x	x	x	
OIS_video	x	x	x	x		x	x		
PFD		x					x	x	
SD	x	x	x	x		x			
SFTD	x	x	x	x		x	x		
VD		x					x	x	
WD	x	x	x	x		x		x	
C_comfort		x		x		x			x
C_braking		x		x		x			x
C_engine		x		x		x			x
C_fuel		x		x		x			x
C_fire		x		x		x			x
C_smoke		x		x		x			x
C_air		x		x		x			x
C_power		x		x		x			x
C_antiice		x		x		x			x

(b) Allocation table (DMM)

Fig. 2.6: A Pareto solution after optimization

Part VII

OUTCOMES AND PERSPECTIVES

CHAPTER 1

OUTCOMES

1.1 Interests and added value

Analyzing the results obtained on the cockpit use-case, the first outcome is that the method permits to identify interesting architectures given a design problem. Indeed, the solutions obtained at the ed of the process are at least as good as those created by a traditional process (except for the *Min. Availability* criterion).

1.1.1 Physical/Functional architectures update

An originality of the method is that, unlike other CDS or design optimization methods, it does not only deal with the definition of a physical architecture but also updates the functional architecture depending on technological choices. This is realized thanks to the *Induced functions* mechanism.

1.1.2 Naturally-bounded design-space

It was stated in Part III, the setting up a the design-space is not straightforward and may lead to ignore some interesting parts of it. Generally, the design-space must be set up explicitly by, for instance, stating the maximum number of components that can be present in the architecture. In some context, that might be useful (e.g. for integration purpose). But most of the time, there is no rationale for these constraints.

Our method does not require the designer to set up bounds for the number of components. Instead, the different algorithms have the freedom to use as many components as they want if no constraints on the maximum number of instances of a component type have been set.

Thanks to the *Minimal chain* rule, the chain search algorithm can be constrained to use only the minimal set of components that it needs to realize the treated function. Consequently, the design-space is naturally bounded by the needs of the functions.

1.1.3 Compatibility with traditional process

The shift from traditional human-based design processes to computer-aided design processes is a major challenge. Nevertheless, an interesting feature of our method is that results of traditional design processes can be used. Indeed, known architectures can be modeled and considered in the initial population of the evolutionary approach. Thus, interesting architectures obtained thanks to a traditional process can be considered as starting points of the solution search and can be compared to new solutions obtained by the optimization method.

1.2 Limitations

As mentioned in the previous Part, a main limitation of the method is that its ability to optimize solutions depends on the considered criteria. Indeed, some criteria like the *Min. Availability* one may require a huge number of iterations to be optimized.

Another limitation of the method deals with the modeling and use of capabilities. Indeed, capabilities are only considered as qualitative and permanent. In some problems, however, capabilities have to be considered in a more complex manner in order to generate valid architectures.

In the last part, main outcomes of the study were presented. Based on these results, some perspectives for future research works can be highlighted.

CHAPTER 2

PERSPECTIVES

2.1 Compatibility rules

2.1.1 Capability performances

In the method, we consider capabilities in a contract-based manner. This relation is only qualitative i.e. the component instances have a capability required by a function. Nothing ensures that:

1. the performances of the components are sufficient to effectively perform the function (initially and in time),
2. the capability is able to be used by several functions.

Taking into account these aspects would restrict the design-space and obtained solutions would be more pertinent. To do so, one should additionally specify:

- the capability performances required by functions,
- the capability performances offered by components,
- the scheduling of functions execution,
- the ability of components capabilities to be shared by different functions at the same time,
- the durability of components capabilities (permanent vs. consumable capabilities).

This would permit to check that, at any time of a given scenario, no component exceeds its performances while executing its allocated functions following their scheduling.

Note that capabilities performances can currently be taken into account during the selection stage. This requires to declare criteria and constraints and to set up evaluators that will compute, based on model data, the violation of the related constraints. The evaluator would certainly require the declaration of a simulation scenario (e.g. pilot inputs in the cockpit use-case).

2.1.2 Other types of rules

Depending on the problem, it might be interesting to consider rules of different types during the generation of architectures. At the moment, the rules are limited to compatibilities between ports and contract-based compatibility between components and functions thanks to the capability concept. Some other constraints such as technological exclusion (i.e. if a technology is used in the architecture, another one cannot be used) or segregation constraints (i.e. two functions cannot be realized by the same component) are not considered at the time being. Nevertheless, many of these constraints that do not require complex evaluations can be easily included in the generation process.

2.1.3 Global initialization

The current initialization process treats solutions sequentially. It might be the case where, because of previously made decisions, no solution can be found to realize function. In this case, the initialization fails and restarts with a new (empty) architecture. Instead, it would be interesting :

- to analyze previous decisions and go back to a state where no inter-blocking is present (back-tracking)
- to solve the problem globally by considering all functions simultaneously.

This would permit to consider constraints impacting several functions but also to improve the performances of the initialization algorithm.

2.2 Performances

In its current version, the algorithm that implements the method takes some seconds to generate and evaluate an alternative. This performances in term of computation time could be improved in several ways. An improvement could permit to generate more alternatives and, consequently, to increase the explored portion of the design-space and the confidence in the optimality of the final selected solutions.

2.2.1 Evolution strategies - Heuristics

The method presented in this thesis is based on evolutionary mechanisms, namely mutation and cross-over. These mechanisms are purely random and do not consider the weaknesses and strength of parent architectures. The idea of evolution strategies is to analyses the parent architecture performances in order to apply to them modifications that will improve the child architecture performances.

Evolution strategies are heuristics. Although some generic strategies may be found, evolution strategies are mostly domain or problem specific and require the elicitation of designers' knowledge. They are also generally related to a particular criterion as it may be difficult to define evolution strategies that optimize conjointly all considered criteria.

For instance, a simple "decrease mass" evolution strategy would consist in analyzing the parent architecture, finding the heaviest chains and trying to replace them by equivalent but lighter ones.

Other evolution strategies could be implemented to improve the optimization of the Availability criterion. A first strategy could consist in a reallocation of functions so that, keeping the same physical architecture, the availability of each function would be maximized. As no other criterion is impacted by allocations, this strategy would have the benefit to create at least as good architectures as their parents. A second strategy could create redundancies in priority on the functions that represent the minimum availability time among all functions. This would mechanically increase the performances of created architectures on the Minimum availability criterion.

The introduction of evolution strategies, in complement to random mechanisms, shall permit to improve the convergence of the optimization process toward the Pareto front of the problem.

2.2.2 Mathematical programming

In our works, we decided to work directly on the model thanks to an algorithm. A idea to improve the process performance would be to transform the model and the rules into a mathematical problem and to use existing mathematical tools to solve it. For instance, constraint programming, a technique that explores branches of a decision tree to find acceptable solutions, could be used to find all solutions to the design problem in a limited time. This approach is used in [Condat, 2011] to allocate functions to components (modules). This technique would only permit to discover viable and valid architectures but not to select the best ones. For this, architectures performances shall be assessed and best architectures shall be selected thanks to selection schemes such as MCDA methods.

Linear programming would permit to optimize solutions by considering linear objectives functions and constraints. This is clearly a limitation as some criteria cannot be assessed by linear models without making simplifying assumptions.

The mathematical model could also be used by an evolutionary optimization algorithm. This approach would present the advantage to be run on different CPUs (parallelization).

2.2.3 Parallelism

Parallelism consists in assigning computation tasks to several cores. These tasks are executed simultaneously. The evaluation of alternatives can be parallelized as an evaluation process do not need data from other processes.

In the same manner, the synthesis and the genetic evolutions of our algorithm could be easily parallelized. This would reduce dramatically the computation time.

For instance the synthesis process for one architecture can take from 5 to 60s and the maximum evaluation time is 0.2s. For 100 architectures, a sequential process would take around 50min whereas a parallel process would take 60s.

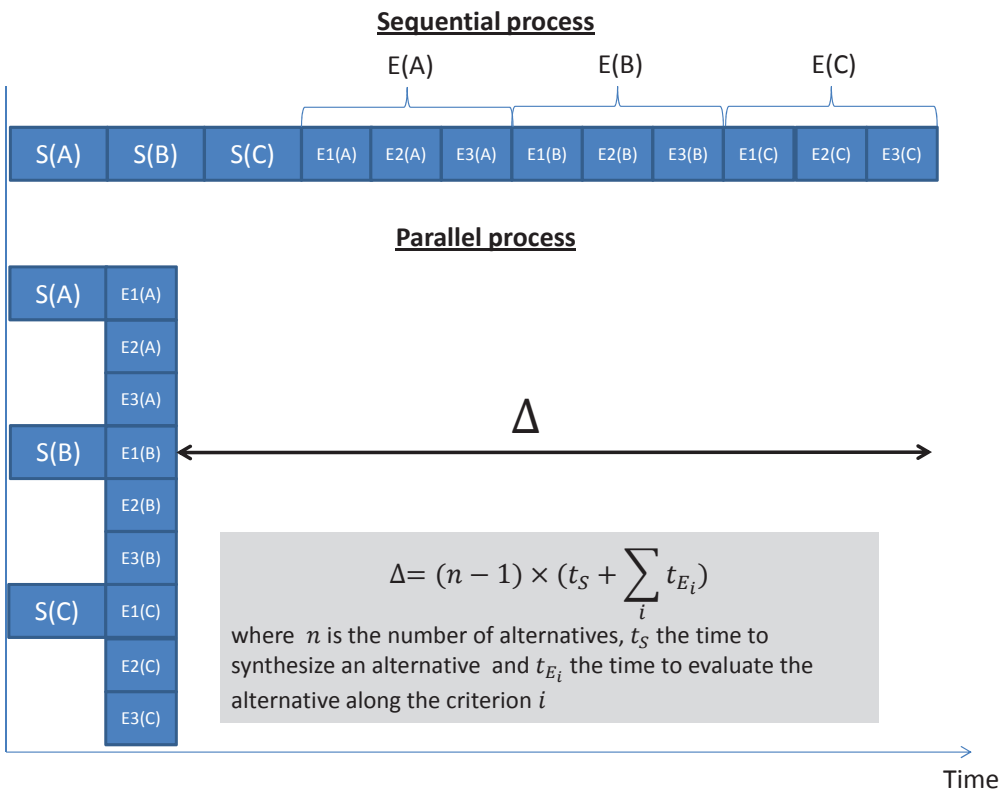


Fig. 2.1: Illustration of sequential vs. parallel processes

2.3 Uncertainties

As the reader will probably notice, uncertainties are not considered in the method although they can play an important role in the decision process. Indeed, during early

design phases, many parameters of the study are uncertain (component characteristics, functional requirements...). This can lead to take decisions that will turn out to be non optimal (or even not adapted) later on, when uncertainties will be more reduced.

The introduction of uncertainties in the method will allow to make robust decisions with respect to the possible evolutions of the design problem parameters. For this, uncertainties must be modeled (distributions, ranges) and considered during the evaluation and selection stages. Uncertainties shall be considered as well during the two generation stages (initialization and evolution) if quantitative capabilities are considered (Section 2.1.1).

2.3.1 Evaluating with uncertainties

Considering uncertainties on the problem parameters, evaluation modules shall not compute a single performance value but rather a performance distribution. The distribution of performances is obtained by propagation of uncertainties or by the Monte-Carlo technique.

The evaluation means may also be uncertain. In order to be used in an evolutionary approach, evaluations should be as quick as possible to limit the runtime of the optimization algorithm. To achieve this, evaluation modules sometimes approximate real performances by using simplified computation algorithms. The metrics produced by these evaluations are thus uncertain. These uncertainties can also be considered by propagation of uncertainties.

2.3.2 Selection with uncertainties

The classical selection processes like the ones used in our method (NSGA-II and NEMO) consider single performance values. Nevertheless, because of uncertainties on the evaluations and on the problem parameters, the selection process must ensure that selected architecture are robustly optimal i.e. that a variation of their parameters in the uncertain domain do not make the selected architectures not optimal. To address this problem, robust optimization (RBO) techniques are emerging (see [Beyer and Sendhoff, 2007] for a review). These techniques include solution robustness measures in the selection of solutions, in addition to optimization objectives.

2.4 Multi-system optimization

The defined method, as almost every CDS methods, only concentrate on one system to be optimized. Nevertheless, as it was previously mentioned, the optimal system is not necessarily the one with the best performances but the one that permits to increase the worth of the upper-level system(s). Because of this, it is important to make the link between

the system optimization process and the upper-system optimization. The objective of this integration would be to drive the lower-level optimization processes depending on the performances of the upper-level system. For instance, the mass criterion for the cockpit as a very low incidence on the mass of the aircraft compared to other systems (propulsion, networks...) so a low effort shall be put in optimizing this criterion. At the contrary, the availability of the cockpit system has a major impact at the aircraft level as, in case of unavailability of a function, the aircraft may not be allowed to take off and may require maintenance. This objective should then be considered as more important than other optimization objectives.

Part VIII

CONCLUSION

The analysis of the industrial context that was performed in this study showed a need of system designers for a method to support them in the architecture design process for complex systems. To address this need, a review of existing computational design methods was performed. This study underlined the weaknesses and limitations of current methods to solve the design problems in the aerospace industry.

A new method for the synthesis and optimization of systems architecture was defined. This method permits to create design alternatives that fulfill some structural rules (namely viability and validity), satisfy some constraints and are pseudo-optimal with respect to some criteria. The method is based on an engineering model formulated on the basis of a meta-model defining the necessary (engineering and optimization) concepts. The model contains all the information of the method :

- the input data of the optimization process (component library, functions, criteria...)
- the data gathered during the optimization process (designers' preferences)
- the generated design alternatives and their performances

The model is also used for evaluation of the different criteria defined in the problem. Evaluation modules gather data from the model to compute architecture performances.

By the use of an evolutionary optimization approach, the method is able to handle large design spaces as it does not try to generate all solutions but rather sets up a smart exploration of the design space. For this reason, the method is particularly suitable in trade studies where a large design-space is considered.

Similar approaches

Many similar approaches are adopted by researchers but also by engineering tool vendors. IBM recently proposed a method to use their tool suite for the optimization of systems architectures [Broodney et al., 2012]. This shows the current interest of such optimization approaches which shall permit to create better products with a reduced development time. This different approaches shall now be tested in parallel to different types of problems (different granularities, design-space sizes...) in order to compare their performances and to determine the optimal scope for each method. Doing so, designers will be able to select, depending on the characteristics of their problems and on the context of the study, the most suitable approach.

The use of a method does not prevent from using other methods in complement. For instance, a method like the one presented in this dissertation can be used in very early design phases in which the considered design-space can be kept open and huge. Given the results of the method, the design-space can be reduced and other methods (e.g. constraint programming or linear programming) can be used on this reduced design-space.

Industrialization

Currently, the architecture optimization methods gain in maturity. The next stage for these methods will be their introduction in the industrial processes. A current brake to this is the culture of designers and engineers, especially in the field of aeronautics. Indeed, they are used to work on the basis of a single main alternative (baseline) and to create new alternatives by modifying some characteristics of it. An important work for researchers is now to demonstrate the interest and the effectiveness of global approaches on real design problems.

Another axis of work for the adoption of these methods will be their tooling. Indeed, in order to be introduced in the industrial processes, the tools used to implement the different methods must be user-friendly and stable. They must also be in line with the different policies and best practices of the company and must present some features such as report generation, exchange with existing databases or use of on line services.

BIBLIOGRAPHY

M. Agarwal and J. Cagan. A blend of different tastes: the language of coffeemakers. *Environment and Planning B*, 25:205–226, 1998.

Air Transport Association of America. ATA Specification 100 - Specification for Manufacturers' Technical Data. Technical report, 1999.

R. Alber and S. Rudolph. On a grammar-based design language that supports automated design generation and creativity. In J. Borg., F. Farrugia, and K. Camilleri, editors, *Knowledge intensive design technology IFIP TC5*, page 19. Springer Netherlands, 2004.

A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. ArcheOpterix : An Extendable Tool for Architecture Optimization of AADL Models. *Evaluation*, 2010.

G. Altshuller. The innovation algorithm: TRIZ, systematic innovation and technical creativity. 1999.

A. Arnold and G. Point. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 1999.

H.-G. Beyer and B. Sendhoff. Robust optimization – A comprehensive survey. *Computer Methods in Applied Mechanics and Engineering*, 196(33-34):3190–3218, July 2007.

R. Bidorff, P. Meyer, and M. Roubens. RUBIS: a bipolar-valued outranking method for the choice problem. *4OR, A Quarterly Journal of Operations Research, Springer-Verlag. (Online*, pages 101007/s10288–007–0045–5pp1–27, 2008.

F. Bolognini, A. Seshia, and K. Shea. A Computational Design Synthesis Method for MEMS Using COMSOL. In *COMSOL Users Conference 2007*, Grenoble, France, 2007.

J. Branke, S. Greco, R. Slowinski, and P. Zielniewicz. Interactive Evolutionary Multiobjective Optimization Using Robust Ordinal Regression. In M. M.Ehrgott, C.M.Fonseca, X.Gandibleux, J.-K.Hao, editor, *Evolutionary Multi-Criterion Optimization (EMO'09)*, volume 24, pages 554–568. LNCS 5467, Springer, Berlin, February 2009.

H. Broodney, D. Dotan, L. Greenberg, and M. Masin. Generic Approach for Systems Design Optimization in MBSE. In *INCOSE Symposium 2012*, Rome, 2012.

C. Bryant, R. Stone, D. McAdams, T. Kurtoglu, and M. Campbell. A computational technique for concept generation. In *Proceedings of IDETC/CIE*, pages 24–28, 2005.

J. Cagan, M. I. Campbell, S. Finger, and T. Tomiyama. A Framework for Computational Design Synthesis: Model and Applications. *Journal of Computing and Information Science in Engineering*, 5(3):171, 2005.

M. Campbell, J. Cagan, and K. Kotovsky. Agent-based synthesis of electromechanical design configurations. *Journal of Mechanical Design*, 122:61, 2000.

H. Condat. *Model-based automatic generation and selection of best architectures.* Master thesis, Master thesis, Technical University of Munich, 2011.

K. Deb. Multi-objective optimization. *Search Methodologies*, pages 273–316, 2005.

K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI*, pages 849–858. Springer, 2000.

D. Dennett. Brain writing and mind reading. *Minnesota Studies inLanguage, Mind and Knowledge,*, 1975.

W. Edwards. How to use multiattribute utility measurement for social decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics*, 1977.

J. Estefan. Survey of model-based systems engineering (MBSE) methodologies. *Incose MBSE Focus Group*, 2007.

FAA. FAA System Safety Handbook. Technical report, 2000.

P. Feiler, D. Gluch, and J. Hudak. The architecture analysis and design language (AADL): An introduction. Technical report, 2006.

J. Figueira, V. Mousseau, and B. Roy. ELECTRE methods. *criteria decision analysis: State of the*, pages 1–35, 2005.

J. Figueira, S. Greco, and R. Slowinski. Building a set of additive value functions representing a reference preorder and intensities of preference: GRIP method. *European Journal of Operational Research*, 195(2):460–486, 2009.

S. Friedenthal and A. Moore. A practical guide to SysML: the systems modeling language. 2011.

P. Fritzon. *Principles of object-oriented modeling and simulation with Modelica 2.1.* 2004.

C. Gorbea, T. Spielmannleitner, U. Lindemann, and E. Fricke. Analysis of Hybrid Vehicle Architectures using Multiple Domain Matrices. In M. Kreimeyer, M., Lindemann, U., Danilovic, editor, *Proceedings of the 10th International DSM Conference*, 2008.

S. Greco, V. Mousseau, and R. Slowinski. Ordinal regression revisited: multiple criteria ranking using a set of additive value functions. *European Journal of Operational Research*, 191(2):416–436, 2008.

M. Gubitosa, J. Anthonis, N. Albarello, and W. Desmet. A system engineering approach for the design optimization of a hydraulic active suspension. In *Vehicle Power and Propulsion Conference, 2009.*, pages 1122–1130. IEEE, 2009.

L. Guerra. Project Life Cycle Module - Space Systems Engineering course. URL `http://spacese.spacegrant.org/`. 2008.

A. Guitouni and J. Martel. Tentative guidelines to help choosing an appropriate MCDA method. *European Journal of Operational Research*, 109(2):501–521, September 1998.

B. Helms and K. Shea. A Framework for Computational Design Synthesis Based on Graph-Grammars and Function-Behavior-Structure. *ASME IDETC*, 2009.

J. Holland. Adaptation in natural and artificial systems. 1975.

V. Holley. *A method to envision highly constrained architectural zones in the design of multi-physics systems for severe operating conditions.* PhD thesis, Ecole Centrale Paris, 2011.

J. Y. Hung and L. F. Gonzalez. On parallel hybrid-electric propulsion system for unmanned aerial vehicles. *Progress in Aerospace Sciences*, pages 1–17, 2012.

IEEE. IEEE 1471. 2007.

INCOSE. *INCOSE Systems Engineering Handbook.* 2007.

Java. Java website. `http://www.java.com/` [Online. Last accessed: 2011-11-06], 2012.

F. Jouault, B.-K. Loukil, and W. Piers. Model-to-model Transformation with ATL. In *Eclipse Conference 2008*, 2008.

A. Kerzhner and C. Paredis. Using Domain Specific Languages to Capture Design Synthesis Knowledge for Model-Based Systems Engineering. In *ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, 2009.

J. Koza, F. B. III, and D. Andre. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. *Intelligence in Design*, 1996.

I. Kroo. Distributed multidisciplinary design and collaborative optimization. *... & Tools for Multicriteria/Multidisciplinary Design*, 2004.

T. Kurtoglu and M. Campbell. Automated synthesis of electromechanical design configurations from empirical analysis of function to form mapping. *Journal of Engineering Design*, 20(1):83–104, 2009.

C. Magee and O. De Weck. Complex system classification. 2004.

M. Masin, A. Sangiovanni-vincentelli, A. Ferrari, L. Mangeruca, H. Broodney, L. Greenberg, M. Sambur, D. Dotan, S. Zolotnizky, and S. Zadorozhniy. Lingua Franca design and integration language (META II final report). Technical report, 2011.

Modelica Association. Modelica Specification 3.2 Rev. 1. Technical report, 2012.

J. Mulvey, R. Vanderbei, and S. Zenios. Robust optimization of large-scale systems. *Operations research*, 1995.

NASA. *NASA System Engineering Handbook*. 2007.

M. Nowostawski. Parallel genetic algorithm taxonomy. *Knowledge-Based Intelligent*, 1999.

Obeo. Acceleo website. `http://www.acceleo.org` [Online. Last accessed: 2011-11-06], 2012.

J. Olvander, B. Lunden, and H. Gavel. A computerized optimization framework for the morphological matrix applied to aircraft conceptual design. *Computer-Aided Design*, 41 (3):187–196, March 2009.

OMG. SysML Specification. 2010.

OMG. OMG Unified Modeling Language (OMG UML) Superstructure 2.4.1. 2011.

OMG. Object Management Group Wiki : Background Material for MBSE Ontology Development. `http://www.omgwiki.org/MBSE/doku.php?id=mbse:background\_material\_for\_mbse\_ontology\_development` [Online. Last accessed: 2011-29-06], 2012.

A. Osborn. Applied imagination. 1953.

R. Rai. Simulation-Based Design of Aircraft Electrical Power Systems. In *Proceedings of the 8th Modelica Conference*, Dresden, 2011.

G. Renner. Genetic algorithms in computer aided design. *Computer-Aided Design*, 35(8): 709–726, July 2003.

S. Robin, K. Trivedi, and A. Puliafito. *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package.* 1996.

B. Rohrbach. Kreativ nach Regeln Methode 635, eine neue Technik zum Losen von Problemen. *Absatzwirtschaft*, 1969.

T. Saaty. Analytic hierarchy process. 1980.

M. Sallak, C. Simon, and J.-F. Aubry. A Reliability graph approach for availability and redundancy allocation: Application to Safety Instrumented Systems. *Ieee Transactions On Reliability*, pages 0–19, 2002.

J.-L. Scharbarg and C. Fraboul. Methods and tools for the temporal analysis of avionic networks. In *New trends in technologies: Control, Management, Computational Intelligence and Network Systems.* 2010.

K. Seo, Z. Fan, J. Hu, and E. Goodman. Toward an automated design method for multi-domain dynamic systems using bond graph and genetic programming. *Mechatronics*, pages 1–21, 2003.

B. Shroer, A. Kain, and U. Lindemann. Supporting creativity in conceptual design: Method 635-extended. *Design Conference*, 2010.

G. Stiny. Shape grammars and the generative specification of painting and sculpture. *Information processing*, 71:125–135, 1972.

Z. Strawbridge, D. D. McAdams, and R. Stone. A computational approach to conceptual design. In *ASME Design Engineering Technical Conference 2002*, Montreal, 2002.

Technische Universitat Munchen. Booggie website. `http://www.booggie.org/` [Online. Last accessed: 2011-11-06], 2012.

K. Ulrich. The role of product architecture in the manufacturing firm. *Research Policy*, 24 (3):419–440, May 1995.

J. Welcomme. MASCODE : un système multi-agent adaptatif pour concevoir des produits complexes . Application à la conception préliminaire avion. 2008.

D. F. Wyatt, D. C. Wynn, J. P. Jarrett, and P. J. Clarkson. Supporting product architecture design using computational design synthesis with network structure constraints. *Research in Engineering Design*, pages 17–52, 2012.

D. Wyatt, D. Wynn, and P. Clarkson. A computational method to support product architecture design. In *Proceedings of ASME-IMECE*, pages 1–14, 2009.

K. Yoon and C. Hwang. Multiple attribute decision making: an introduction. 1995.

E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm. 2001.

F. Zwicky. The morphological method of analysis and construction. 1948.

LIST OF FIGURES

2.1 The research context 18

1.1 Hierarchy in systems 23

1.2 Example of physical architectures for the propulsion system of an aircraft (adapted from Hung and Gonzalez [2012]) 25

2.1 Development processes in the aerospace industry 28

2.1 Illustration of Cagan et al. framework 37

2.2 A morphological matrix for the motorcycle design problem (from Olvander et al. [2009]) 38

2.3 Example of component DSM, function DSM and DMM (from [Gorbea et al., 2008]) 39

2.4 Example of diagram in SysML 41

2.5 Example of a graph-grammar for the design of truss-like structures (from [Alber and Rudolph, 2004]) 43

2.6 An hypothetical function to component rule (from [Kurtoglu and Campbell, 2009]) 45

2.7 Example of an electrical circuit and its program tree (from [Koza et al., 1996]) 48

2.8 The guidance process 51

2.1 Meta-model, model and language (from [OMG, 2012]) 62

2.2 Meta-model overview 63

3.1 Functional architecture 72

3.2 Component types 73

3.3 System 73

3.4 Treatment of $F1$ 74

3.5 Architectures after $F1$ treatment for the two possible chains 74
3.6 Architecture after $F1$ treatment 75
3.7 Treatment of $F2$ 75
3.8 Architectures after $F2$ treatment for the two possible chains 76
3.9 Architecture after treatment of $F3$ 76
3.10 Final architecture 77
3.11 Different possibly generated architectures 77
3.12 Architecture after the removal of chains ($F2$,$F3$ and $FI3_1$) 80
3.13 Architecture after simple mutation 80
3.14 Crossover 83

4.1 A Simulink model 85
4.2 On-demand evaluations of performances 86

2.1 The SAMOA architecture 96

1.1 A380 cockpit 107
1.2 Cockpit interfaces 108

2.1 Navigation Display 113
2.2 Anti-Ice control function 115
2.3 Reference architecture 119
2.4 A Pareto solution after initialization (reuse probability = 1.0) 123
2.5 Effect of the redundancy creation operator on the availability of the C_fire function 127
2.6 A Pareto solution after optimization 129

2.1 Illustration of sequential vs. parallel processes 137

A.1 Display Unit (DU) 155
A.2 Shared Display Unit (sDU) 155
A.3 Private Display Unit (pDU) 155
A.4 Shared Dumb Display Unit (sDDU) 156

A.5 Private Dumb Display Unit (pDDU) 156
A.6 Private Smart Display Unit (pSDU) 156
A.7 Shared Smart Display Unit (sSDU) 157
A.8 Head-Up Display (HUD) 157
A.9 Central Processing Unit (CPU) 158
A.10 Graphical Processing Unit (GPU) 158
A.11 Integrated Control Panel (ICP) 158
A.12 Cable 159
A.13 Data Cable (dC) 159
A.14 Electrical Cable (eC) 159
A.15 Network 160
A.16 Controller Area Network (CAN) 160
A.17 Advanced Data Communication Network (ADCN) 160

B.1 Fault tree of display functions using DDU 162
B.2 Fault tree of display functions using HUD 163
B.3 Fault tree of control functions 164

LIST OF TABLES

2.1 CDS methods classification in the CDS framework (1 of 2) 55
2.2 CDS methods classification in the CDS framework (2 of 2) 56

2.1 List of Display functions 114
2.2 List of Control functions 115
2.3 List of component types 116
2.4 Performances of the reference architecture 120
2.5 Pareto solutions after population initialization (reuse probability = 0.0) . . 121
2.6 Pareto solutions after population initialization (reuse probability = 1.0) . . 122
2.7 Parameters of the first run (mutation only) 124
2.8 Pareto solutions after optimization (Mutation) 125
2.9 Parameters of the second run (mutation + redundancy creation) 125
2.10 Pareto solutions after optimization (Mutation + Redundancy creation) . . . 126
2.11 Pareto solutions after optimization (Crossover + Mutation + Redundancy creation) 128

Appendix A

DESCRIPTION OF COMPONENT TYPES

In the following, component types considered for the cockpit study are described in details. Data were modified for confidentiality reasons.

Information in italics are inherited informations.

Port	
Name	in_port
Direction	IN
Type	-
Multiplicity	-
Attributes	-
Flows	-

ComponentType	
Name	DU
Basetype	-
Abstract	Yes
Max instances	6
Capabilities	-
Attributes	-
Induced functions	-

Port	
Name	out_port
Direction	OUT
Type	-
Multiplicity	-
Attributes	-
Flows	-

Fig. A.1: Display Unit (DU)

Port	
Name	*in_port*
Direction	*IN*
Type	-
Multiplicity	Infinity
Attributes	-
Flows	-

ComponentType	
Name	sDU
Basetype	DU
Abstract	Yes
Max instances	2
Capabilities	-
Attributes	-
Induced functions	-

Port	
Name	*out_port*
Direction	*OUT*
Type	-
Multiplicity	Infinity
Attributes	-
Flows	-

Fig. A.2: Shared Display Unit (sDU)

Port	
Name	*in_port*
Direction	*IN*
Type	-
Multiplicity	1
Attributes	-
Flows	-

ComponentType	
Name	pDU
Basetype	DU
Abstract	Yes
Max instances	6
Capabilities	-
Attributes	-
Induced functions	-

Port	
Name	*out_port*
Direction	*OUT*
Type	-
Multiplicity	1
Attributes	-
Flows	-

Fig. A.3: Private Display Unit (pDU)

Port		
Name	*in_port*	
Direction	*IN*	
Type	Video	
Multiplicity	*Infinity*	
Attributes	portType	female
Flows	-	

ComponentType		
Name	sDDU	
Basetype	sDU	
Abstract	No	
Max instances	6	
Capabilities	-	
Attributes	Location	Cockpit
	Mass	5,1
	Conso	52,0
	MTTF	64900

Port		
Name	*out_port*	
Direction	*OUT*	
Type	Image	
Multiplicity	*Infinity*	
Attributes	portType	female
Flows	-	

Fig. A.4: Shared Dumb Display Unit (sDDU)

Port		
Name	*in_port*	
Direction	*IN*	
Type	Video	
Multiplicity	*Infinity*	
Attributes	portType	female
Flows	-	

ComponentType		
Name	pDDU	
Basetype	pDU	
Abstract	No	
Max instances	6	
Capabilities	Perso_centered	
Attributes	Location	Cockpit
	Mass	5,1
	Conso	52,0
	MTTF	64900

Port		
Name	*out_port*	
Direction	*OUT*	
Type	Image	
Multiplicity	*1*	
Attributes	portType	female
Flows	-	

Fig. A.5: Private Dumb Display Unit (pDDU)

Port		
Name	*in_port*	
Direction	*IN*	
Type	Video	
Multiplicity	*Infinity*	
Attributes	portType	female
Flows	-	

ComponentType		
Name	pSDU	
Basetype	pDU	
Abstract	No	
Max instances	6	
Capabilities	Computing Graphical Perso_centered	
Attributes	Location	Cockpit
	Mass	6,4
	Conso	95,0
	MTTF	38000

Port		
Name	*out_port*	
Direction	*OUT*	
Type	Image	
Multiplicity	*1*	
Attributes	portType	female
Flows	-	

Fig. A.6: Private Smart Display Unit (pSDU)

ComponentType

Name	sSDU	
Basetype	sDU	
Abstract	No	
Max instances	6	
Capabilities	Computing Graphical	
Attributes	Location	Cockpit
	Mass	6,4
	Conso	95,0
	MTTF	38000

Port

Name	*in_port*	
Direction	*IN*	
Type	V	
Multiplicity	*Infinity*	
Attributes	portType	female
Flows	-	

Port

Name	*out_port*	
Direction	*OUT*	
Type	I	
Multiplicity	*Infinity*	
Attributes	portType	female
Flows	-	

Fig. A.7: Shared Smart Display Unit (sSDU)

ComponentType

Name	HUD	
Basetype	-	
Abstract	No	
Max instances	2	
Capabilities	-	
Attributes	Location	Cockpit
	Mass	6,4
	Conso	95,0
	MTTF	38000

Port

Name	in_port	
Direction	IN	
Type	Video	
Multiplicity	2	
Attributes	portType	female
Flows	-	

Port

Name	out_port	
Direction	OUT	
Type	Image	
Multiplicity	1	
Attributes	portType	female
Flows	-	

Function

Name	Control
Inputs	c (Control)
Outputs	cd (Data)
Capabilities	-
Attributes	-

Port

Name	control_port	
Direction	IN	
Type	Data	
Multiplicity	1	
Attributes	portType	female
Flows	cd (Data)	

Fig. A.8: Head-Up Display (HUD)

Port		
Name	in_port	
Direction	IN	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	female
Flows	-	

ComponentType		
Name	CPU	
Basetype	-	
Abstract	No	
Max instances	-	
Capabilities	Computing	
Attributes	Location	Bay
	Mass	0,8
	Conso	59,0
	MTTF	100000

Port		
Name	out_port	
Direction	OUT	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	female
Flows	-	

Fig. A.9: Central Processing Unit (CPU)

Port		
Name	in_port	
Direction	IN	
Type	VideoData	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

ComponentType		
Name	GPU	
Basetype	-	
Abstract	No	
Max instances	-	
Capabilities	Graphical	
Attributes	Location	Bay
	Mass	0,8
	Conso	59,0
	MTTF	100000

Port		
Name	out_port	
Direction	OUT	
Type	Video	
Multiplicity	2	
Attributes	portType	female
Flows	-	

Fig. A.10: Graphical Processing Unit (GPU)

Port		
Name	in_port	
Direction	IN	
Type	Control	
Multiplicity	Infinity	
Attributes	portType	female
Flows	-	

ComponentType		
Name	ICP	
Basetype	-	
Abstract	No	
Max instances	-	
Capabilities	-	
Attributes	Location	Cockpit
	Mass	4,0
	Conso	1,0
	MTTF	40000

Port		
Name	out_port	
Direction	OUT	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	female
Flows	-	

Fig. A.11: Integrated Control Panel (ICP)

Port		
Name	in_port	
Direction	IN	
Type	-	
Multiplicity	1	
Attributes	portType	male
Flows	-	

ComponentType		
Name	Cable	
Basetype	-	
Abstract	Yes	
Max instances	-	
Capabilities	-	
Attributes	-	

Port		
Name	out_port	
Direction	OUT	
Type	-	
Multiplicity	1	
Attributes	portType	male
Flows	-	

Fig. A.12: Cable

Port		
Name	in_port	
Direction	*IN*	
Type	Data	
Multiplicity	*1*	
Attributes	*portType*	*male*
Flows	-	

ComponentType		
Name	dC	
Basetype	Cable	
Abstract	No	
Max instances	-	
Capabilities	-	
Attributes	Linear_mass	$8{,}15.10^{-3}$

Port		
Name	out_port	
Direction	*OUT*	
Type	Data	
Multiplicity	*1*	
Attributes	*portType*	*male*
Flows	-	

Fig. A.13: Data Cable (dC)

Port		
Name	in_port	
Direction	*IN*	
Type	Elec	
Multiplicity	*1*	
Attributes	*portType*	*male*
Flows	-	

ComponentType		
Name	dC	
Basetype	Cable	
Abstract	No	
Max instances	-	
Capabilities	-	
Attributes	Linear_mass	$4{,}0.10^{-3}$

Port		
Name	out_port	
Direction	*OUT*	
Type	Elec	
Multiplicity	*1*	
Attributes	*portType*	*male*
Flows	-	

Fig. A.14: Electrical Cable (eC)

Port		
Name	in_port	
Direction	IN	
Type	-	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

ComponentType		
Name	Network	
Basetype	-	
Abstract	Yes	
Max instances	-	
Capabilities	-	
Attributes	-	

Port		
Name	out_port	
Direction	OUT	
Type	-	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

Fig. A.15: Network

Port		
Name	in_port	
Direction	IN	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

ComponentType		
Name	CAN	
Basetype	-	
Abstract	Yes	
Max instances	-	
Capabilities	-	
Attributes	Linear_mass	$1{,}7.10^{-3}$
	MTTF	1020000

Port		
Name	out_port	
Direction	OUT	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

Fig. A.16: Controller Area Network (CAN)

Port		
Name	in_port	
Direction	IN	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

ComponentType		
Name	ADCN	
Basetype	-	
Abstract	Yes	
Max instances	-	
Capabilities	-	
Attributes	Linear_mass	0,0
	MTTF	250000

Port		
Name	out_port	
Direction	OUT	
Type	Data	
Multiplicity	Infinity	
Attributes	portType	male
Flows	-	

Fig. A.17: Advanced Data Communication Network (ADCN)

Appendix B

AVAILABILITY ANALYSIS OF THE REFERENCE ARCHITECTURE

This Appendix presents the results of the Fault Tree Analysis (FTA) conducted on the Reference architecture. The Availability time A of each function is computed thanks to the analysis of the architecture and the building of a fault tree which decomposes the failure of functions into failure of components.

The way functions are realized are very similar so only 3 FTA have to be conducted : display function using a Dumb Display Unit (DDU), display function using a Head-Up Display (HUD) and control function.

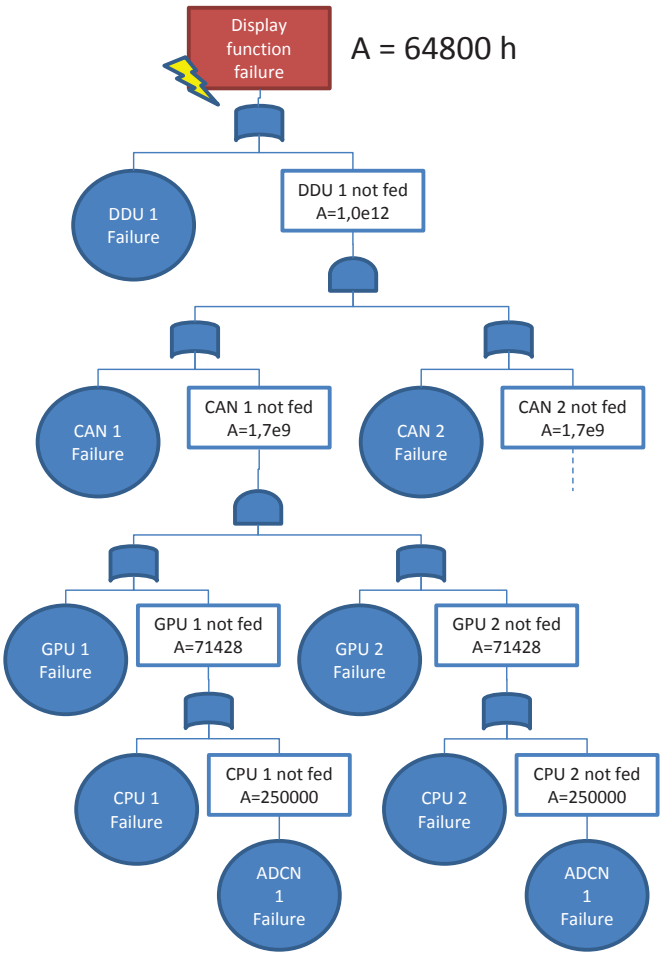


Fig. B.1: Fault tree of display functions using DDU

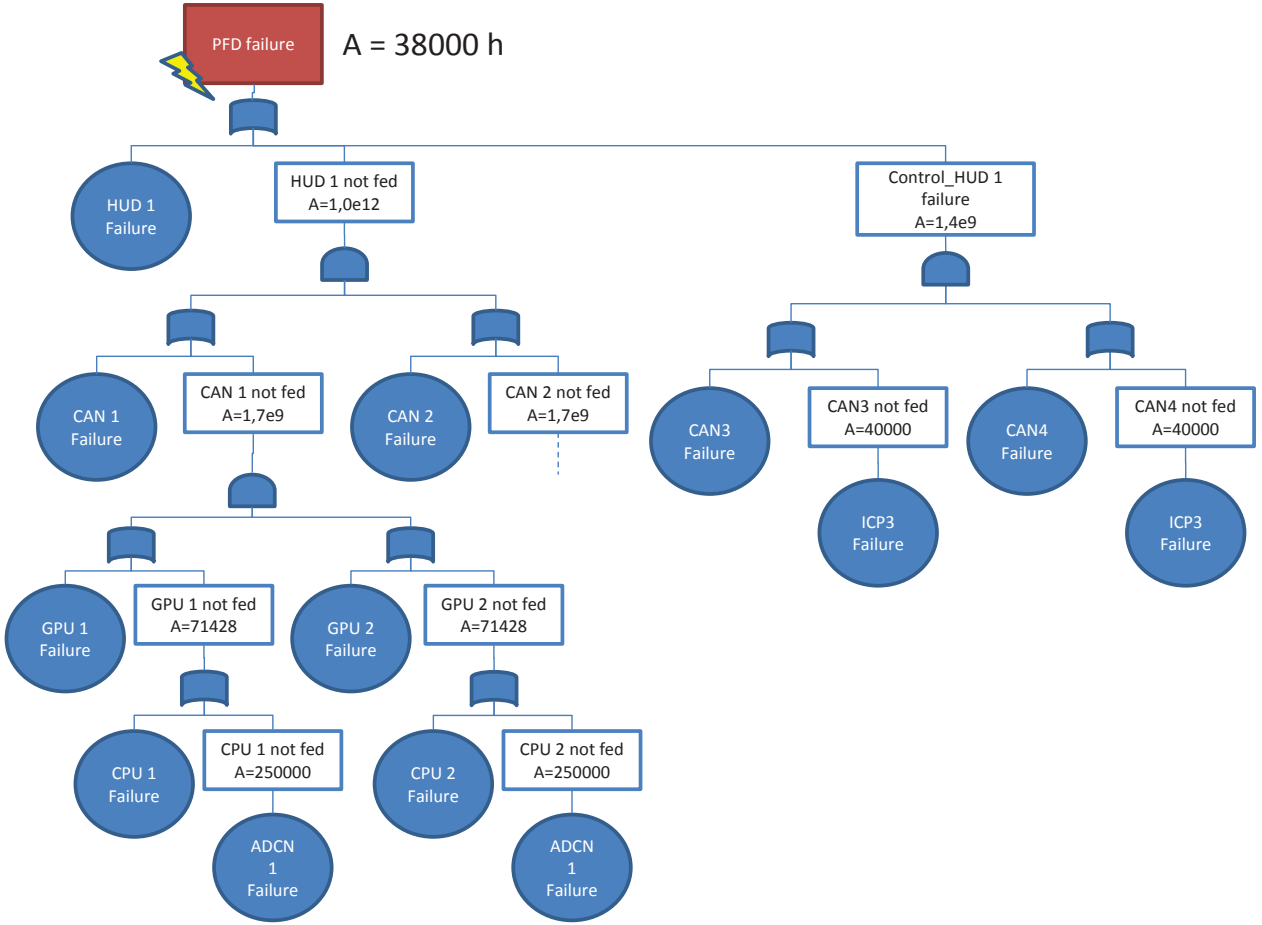


Fig. B.2: Fault tree of display functions using HUD

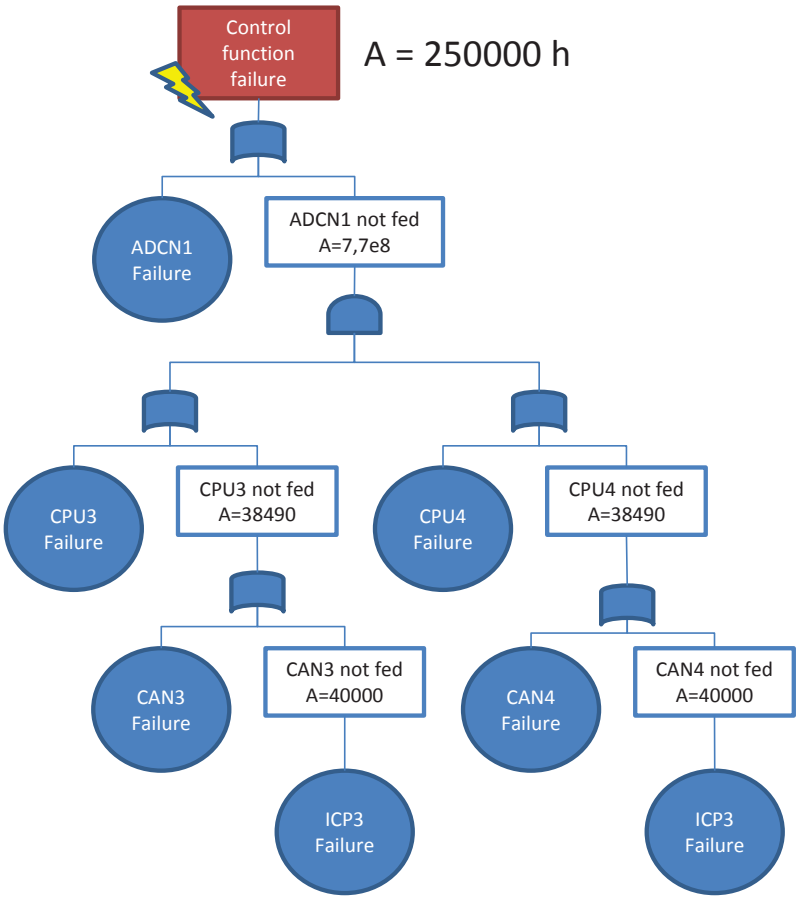


Fig. B.3: Fault tree of control functions

Appendix C

HOW TO USE THE SAMOA PROFILE

The profile was developed using the IBM Rational Rhapsody 7.6.1 tool. Some constructs may be dependent on this tool. Also, the parser uses the Rhapsody API.

Initiating the project

1. Create a new SysML model
2. Import the SAMOA profile (File > Add Profile To Model > SAMOA_Porfile.sbs > Add as Reference)
3. Create a "Problem Definition" package

Modeling the problem

1. Types

1. Create a BDD for the definition of types
2. Create a block

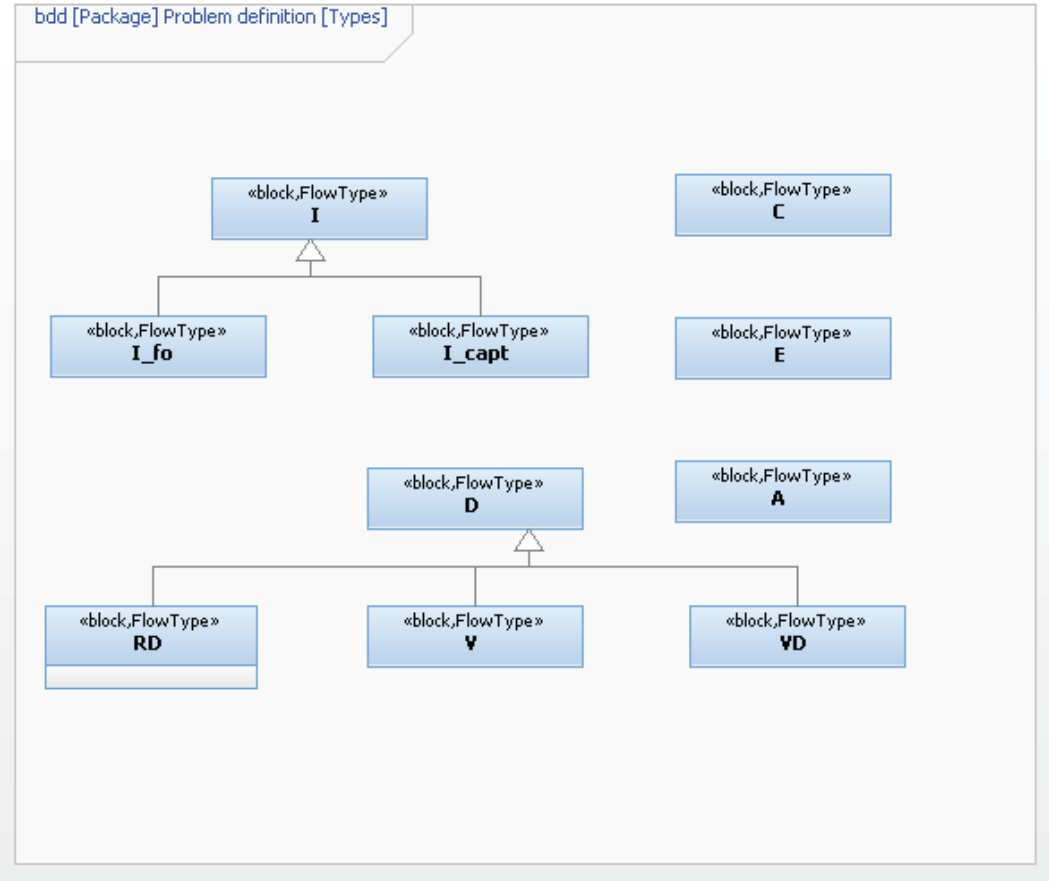


Figure 1 - Block Definition Diagram of Types

2. System

1. Create a block called "System"
2. Set the stereotype of the block as "Architecture"
3. Create an IBD of System and populate it with ports (see 3)

Figure 2 - Internal Block Diagram of System

3. Ports

1. Create a flow port
2. Set the type and multiplicity
3. Add attributes as Tags
4. Add flows by creating dependencies to corresponding pins

Note: The relation to Pins can, in theory, be done by adding Pins as Tags but the Pins are not accessible in the Rhapsody version we used.

4. Functions

1. Create an activity diagram for System
2. Add an action
3. Add pins for inputs and outputs
4. Set the type of pins
5. Add capabilities (see 5)
6. Add attributes as Tags

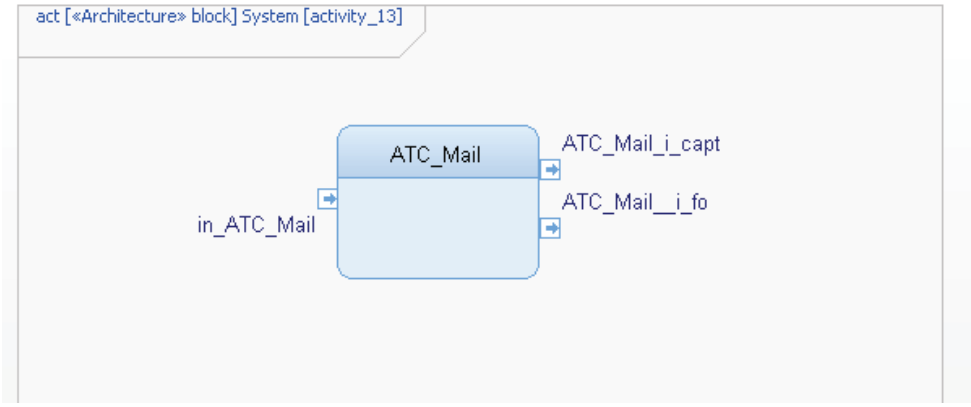


Figure 3 - Activity Diagram of System

5. Capabilities

1. Add a Tag to the component type or to the function
2. Set the stereotype of the Tag as “Capability”

Or (reuse)

1. Locate the existing capability Tag
2. Create a dependency link from the component type or function to the Tag

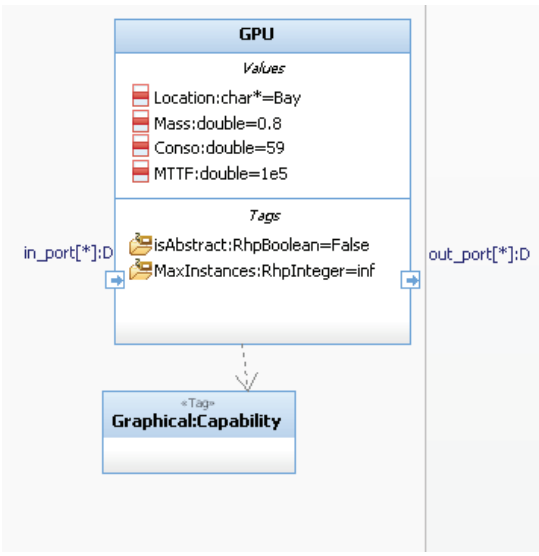


Figure 4 - A component type reusing an existing capability

6. Component types

1. Create a BDD
2. Create a block
3. Set the stereotype of the block as “ComponentType”

4. Add ports (see 3)
5. Add attributes as block Attributes
6. Set maxInstances and isAbstract tags
7. Add Capabilities (see 5)

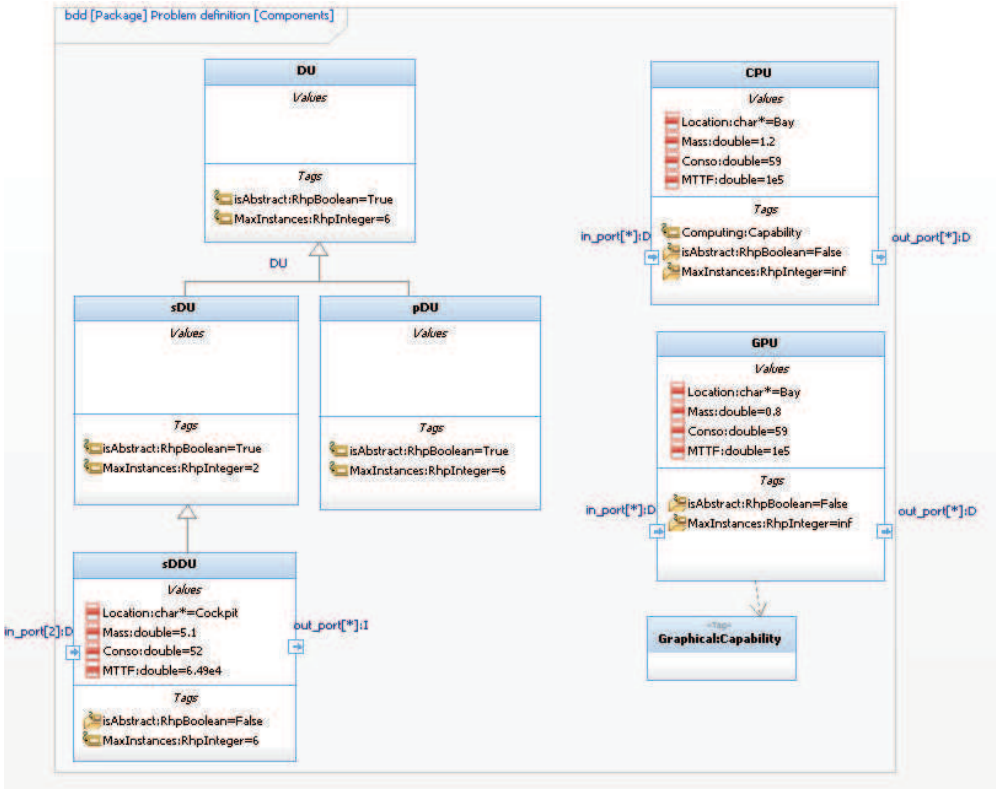


Figure 5 - Block Definition Diagram for Component types

7. Criteria

1. Create a BDD
2. Create a block
3. Set the stereotype of the block as “Criterion”
4. Set the value of the “evaluationMethod” tag

Note: The evaluation method tag corresponds to the name of the method contained in the EvaluationMethods class of the Java project (SAMOA_v2 > src > Modules > EvaluationMethods).

8. Objectives

1. Create a block
2. Set the stereotype of the block as “Objective”
3. Set the value of the “Direction” tag
4. Create an Association relation between the objective and its criterion

9. Constraints

1. Create a block
2. Set the stereotype of the block as “Constraint”
3. Set the values of the “Direction”, “Threshold” and “Criticality” tags
4. Create an Association relation between the objective and its criterion

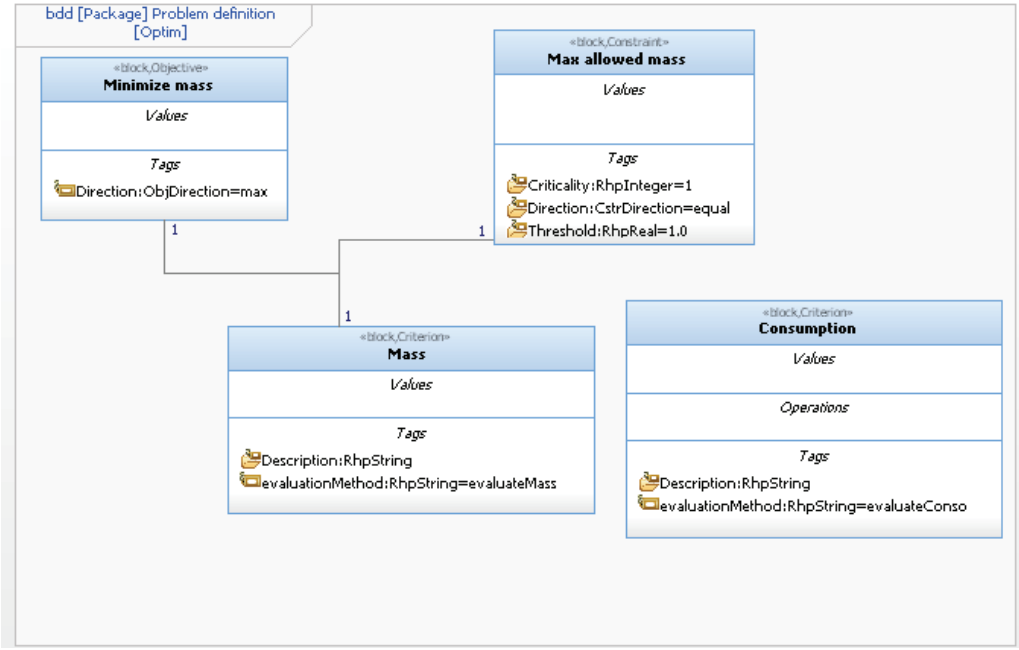


Figure 6 - Block Definition Diagram for Criteria, Objectives and Constraints