THESIS

presented at

Université Paul Sabatier - Toulouse III

U.F.R. MATHÉMATIQUES, INFORMATIQUE ET GESTION

to obtain the title of

DOCTEUR DE L'UNIVERSITÉ DE TOULOUSE

delivered by

UNIVERSITÉ PAUL SABATIER - TOULOUSE III

Mention INFORMATIQUE

by

TOM JORQUERA

Doctoral school: Informatique et Télécommunication
Laboratory: Institut de Recherche en Informatique de Toulouse
Team: Systèmes Multi-Agents Coopératifs

An Adaptive Multi-Agent System for Self-Organizing Continuous Optimization

Tom Jorquera

AN ADAPTIVE MULTI-AGENT SYSTEM FOR SELF-ORGANIZING CONTINUOUS OPTIMIZATION

Supervisors: Marie-Pierre GLEIZES, Jean-Pierre GEORGÉ
Université de Toulouse III

Abstract

This thesis presents a novel approach to distribute complex continuous optimization processes among a network of cooperative agents. Continuous optimization is a very broad field, including multiple specialized sub-domains aiming at efficiently solving a specific subset of continuous optimization problems. While this approach has proven successful for multiple application domains, it has shown its limitations on highly complex optimization problems, such as complex system design optimization. This kind of problems usually involves a large number of heterogeneous models coming from several interdependent disciplines.

In an effort to tackle such complex problems, the field of multidisciplinary optimization methods was proposed. Multidisciplinary optimization methods propose to distribute the optimization process, often by reformulating the original problem is a way that reduce the interconnections between the disciplines. However these methods present several drawbacks regarding the difficulty to correctly apply them, as well as their lack of flexibility.

Using the AMAS (Adaptive Multi-Agent Systems) theory, we propose a multi-agent system which distributes the optimization process, applying local optimizations on the different parts of the problem while maintaining a consistent global state. The AMAS theory, developed by the SMAC team, focuses on cooperation as the fundamental mechanism for the design of complex artificial systems. The theory advocates the design of self-adaptive agents, interacting cooperatively at a local level in order to help each others to attain their local goals.

Based on the AMAS theory, we propose a general agent-based representation of continuous optimization problems. From this representation we propose a nominal behavior for the agents in order to do the optimization process. We then identify some specific configurations which would disturb this nominal optimization process, and present a set of cooperative behaviors for the agents to identify and solve these problematic configurations.

At last, we use the cooperation mechanisms we introduced as the basis for more general Collective Problem Solving Patterns. These patterns are high-level guideline to identify and solve potential problematic configurations which can arise in distributed problem solving systems. They provide a specific cooperative mechanism for the agents, using abstract indicators that are to be instantiated on the problem at hand.

We validate our system on multiple test cases, using well-known classical optimization problems as well as multidisciplinary optimization benchmarks. In order to study the scalability properties of our system, we proposed two different ways to automatically generate valid optimization problems. Using these techniques we generate large test sets which allow us to validate several properties of our system.

Summary

Introduction 1

I Context of the Study and State of the Art 5

1 Continuous Optimization 7

2 Multi-Objective Optimization 23

3 Multidisciplinary Optimization 31

4 Multi-Agent Systems for Optimization and the AMAS Theory 51

II A Multi-Agent System for Continuous Optimization 63

5 Agent-Based Modeling and Simulation of a Continuous Optimization Problem 67

6 Agents Behavior 77

7 Extending the Multi-Agent System for Uncertainties 111

III Design, Implementation and Extending the AMAS4Opt Building Blocks 119

8 ADELFE and MAY Architecture 123

9 Collective Problem Solving Patterns 137

IV Experiments and Validation 147

10 Behavior Validation using Academic Test Cases 151

11 Comparison with Existing Methods 161

12 Evaluating Scalability Performances using Generated Test Cases 167

Conclusion and Perspectives 177

V Appendix 181

Author's Bibliography 183

Bibliography 185

Contents

Introduction **1**

I Context of the Study and State of the Art **5**

1 Continuous Optimization **7**
1.1 Basic Concepts 7
1.1.1 Continuous versus Discrete Optimization 9
1.1.2 No Free Lunch Theorem 10
1.2 Continuous Optimization Methods 11
1.3 Linear Optimization 12
1.4 Local Methods 13
1.4.1 Using Derivatives 13
1.4.2 Derivative-Free 15
1.5 Global Methods 16
1.5.1 Exact Methods 17
1.5.2 Heuristic Methods 17
1.6 Analysis of Continuous Optimization 20

2 Multi-Objective Optimization **23**
2.1 A Priori Methods 24
2.1.1 Objectives Aggregation 24
2.1.2 Lexicographic Method 25
2.1.3 ϵ-constraint Method 25
2.1.4 Goal Programming 25
2.1.5 MinMax Method 26
2.1.6 Analysis of a priori methods 26
2.2 A Posteriori Methods 26
2.2.1 Pareto Dominance 27
2.2.2 Multi-Objective Evolutionary Algorithms 28
2.3 Analysis of MOO 29

3 Multidisciplinary Optimization **31**
3.1 Mono-Level Methods 33
3.1.1 Multidisciplinary Feasible 33
3.1.2 Individual Discipline Feasible 34
3.1.3 All-At-Once 35
3.2 Multi-Level Methods 35

3.2.1 Concurrent Subspace Optimization 35
3.2.2 Collaborative Optimization 36
3.2.3 Bilevel Integrated System Synthesis 37
3.2.4 Asymmetric Subspace Optimization 37
3.2.5 MDO based on Independent Subspaces 38
3.2.6 Quasiseparable Subsystems Decomposition 38
3.3 Design Optimization Under Uncertainties 39
3.3.1 Several types of uncertainties 39
3.3.2 Uncertainties Modeling Techniques 40
3.3.3 Using Uncertainty for Robust Optimization 43
3.3.4 Uncertainties in Multidisciplinary Optimization 45
3.4 Analysis of MDO 45

4 Multi-Agent Systems for Optimization and the AMAS Theory 51
4.1 Multi-Agent Systems 51
4.1.1 Principles of Multi-Agent System 52
4.1.2 Self-* capabilities 52
4.1.3 Multi-Agent Systems for Distributed Problem Solving 53
4.2 The Adaptive Multi-Agent Systems Theory 57
4.2.1 Theorem of Functional Adequacy 57
4.2.2 Cooperative Agents and Non Cooperative Situations 59
4.2.3 The Importance of Locality 60
4.2.4 ADELFE - A Method for Designing AMAS 61
4.2.5 Conclusion on the Adaptive Multi-Agent Systems Theory 61

II A Multi-Agent System for Continuous Optimization 63

5 Agent-Based Modeling and Simulation of a Continuous Optimization Problem 67
5.1 NDMO: A Natural Domain Modeling for Optimization 67
5.1.1 Models 69
5.1.2 Design Variables 69
5.1.3 Outputs 69
5.1.4 Constraints 69
5.1.5 Objectives 71
5.2 From an Optimization Problem to a Multi-Agent System 71
5.2.1 Model Agent 72
5.2.2 Variable Agent 72
5.2.3 Output Agent 73
5.2.4 Constraint Agent 74
5.2.5 Objective Agent 74

6 Agents Behavior 77
6.1 Problem Simulation 77
6.1.1 Variable Agent 78
6.1.2 Model Agent 79
6.1.3 Output Agent 80

6.1.4 Constraint/Objective Agent . . . 80
6.2 Collective Solving . . . 81
6.2.1 Variable Agent . . . 82
6.2.2 Model Agent . . . 84
6.2.3 Output Agent . . . 85
6.2.4 Constraint Agent . . . 85
6.2.5 Objective Agent . . . 87
6.2.6 Adaptive Agents and Co-design . . . 88
6.3 Non-Cooperative Situations . . . 90
6.3.1 Conflicting Requests . . . 91
6.3.2 Cooperative Trajectories . . . 95
6.3.3 Cycle Solving . . . 99
6.3.4 Hidden Dependencies . . . 101
6.3.5 Asynchronous Messages . . . 105
6.3.6 Summary of Non-Cooperative Situations . . . 107

7 Extending the Multi-Agent System for Uncertainties 111
7.1 From Deterministic Optimization to Optimization under Uncertainties . . . 111
7.2 Manipulating Uncertain Values . . . 112
7.3 Uncertainties Propagators . . . 114
7.4 Conclusion on Uncertainties Handling . . . 115

III Design, Implementation and Extending the AMAS4Opt Building Blocks 119

8 ADELFE and MAY Architecture 123
8.1 Overview of ADELFE . . . 123
8.2 Applying ADELFE for the Design of a Continuous Optimization Tool . . . 124
8.3 MAY Agent Architecture . . . 126
8.3.1 Behavior . . . 127
8.3.2 Communication . . . 129
8.3.3 Monitoring . . . 131
8.4 MAY MAS Architecture . . . 131
8.5 Integration into the Prototype . . . 134
8.5.1 MAS . . . 134
8.5.2 CORE . . . 134
8.5.3 GUI . . . 134

9 Collective Problem Solving Patterns 137
9.1 Introduction - Collective Problem Solving Patterns are not Design Patterns . . 138
9.2 Description of a Problem Solving Pattern . . . 139
9.2.1 Agent Roles . . . 139
9.2.2 Pattern Description . . . 140
9.3 Identified Collective Problem Solving Patterns . . . 140
9.3.1 Conflicting Requests . . . 140
9.3.2 Cooperative Trajectories . . . 142

9.3.3 Cycle Solving 142
9.3.4 Hidden Dependencies 142
9.3.5 Asynchronous Requests 143
9.4 Conclusion on Collective Problem Solving Patterns 144

IV Experiments and Validation **147**

10 Behavior Validation using Academic Test Cases **151**
10.1 Turbofan Problem 151
10.2 Viennet1 151
10.3 Rosenbrock's valley 152
10.4 Alexandrov Problem 153
10.5 Analysis of Academic Test Cases 154
10.6 Optimization under Uncertainties 156
10.7 Adaptation to Perturbations 157
10.7.1 Perturbated Alexandrov Problem 157
10.7.2 Perturbated Turbofan Problem 157

11 Comparison with Existing Methods **161**
11.1 Comparison Criteria 161
11.2 Comparison Problem 1 162
11.3 Comparison Problem 2 164
11.4 Comparison Synthesis 165

12 Evaluating Scalability Performances using Generated Test Cases **167**
12.1 Generated Problem Graphs 167
12.1.1 Generating NDMO Agent Graphs 167
12.1.2 Experimental Results 169
12.1.3 Analysis of Performances 171
12.2 Springs Networks 172
12.2.1 Representing Springs Networks with NDMO 172
12.2.2 Springs Networks Experiments 173

Conclusion and Perspectives **177**

V Appendix **181**

Author's Bibliography **183**

Bibliography **185**

Introduction

Complex Continuous Optimization and Multi-Agent Systems

Continuous optimization is a very large field, including various methods tailored for diverse but specific requirements. While this approach was successful in providing a toolbox of specialized methods, the evolution of industrial needs draws attention to some of its limitations. Indeed, current optimization methods fail to handle the more complex optimization problems. These problems are characterized by heavy calculus, many interdependencies between their components and the expertise domains they involve. Classical optimization tools struggle with these problems because of these factors, and specific methods have been proposed to handle this complexity, giving birth to the field of Multidisciplinary Design Optimization (MDO). However, MDO methods involve possibly important transformations to the original problem in order to divide it into simpler ones, which makes these approaches somewhat cumbersome and potentially inefficient for highly connected problems.
This issue is especially present in the context of complex, real-world systems design (aircrafts, space shuttles *etc.*), where the complexity of the problem is usually a reflection of the complexity of the system being built. In this context, the problem is not only very large and interconnected but also often not completely defined and is continuously corrected and modified during the design process. The optimization problem is basically in a feedback loop with the designer solving it, where a solution to the problem provides new information to the designer, which in return refines the problem formulation and so on. Existing methods are not adapted to such a dynamic co-design process, as they often need to be restarted from scratch if the problem formulation is modified.

At the same time, new paradigms are being proposed to handle systemic complexity. One of the most successful is the field of Multi-Agent Systems (MAS). This approach proposes to handle problem complexity using systems of autonomous interacting agents. Instead of reducing the problem in order to solve it using a centralized process, MAS techniques preserve the original problem and use decentralized mechanisms in order to spread the solving effort among the agents. MAS has proved successful in the field of combinatorial optimization, on problems such as graph coloring, sensors network or scheduling. However, very few applications of MAS have been proposed in the context of continuous optimization, and these proposals are restricted to specific application domains or optimization categories.

During the last ten years, the scientific community has pursued the effort to bridge the gap between these two apparently irreconcilable approaches: mathematical optimization and MAS. The goal of this thesis is to contribute to this effort by addressing this mostly unexplored potential application field of MAS: complex continuous optimization.

Contributions of the Thesis

The first contributions of this thesis concern the design of a MAS for the solving of complex continuous optimization problems:

We study continuous optimization problems and show how all of them share a common structure. Using this observation, **we propose a representation of continuous optimization problems using entities graphs**, which we call *Natural Domain Modeling for Optimization* (NDMO), as it preserves the natural expression of the problem.

Based on the NDMO representation **we identify several agent roles for the graph entities. For each agent role we propose a nominal behavior in order to produce a MAS capable of distributing the optimization process**. This system is not only able to distribute the continuous optimization process, but is also capable of adapting to changes made by the expert to the problem formulation *during solving*, enabling the co-design of a system.

In accordance with the Adaptive Multi-Agent Systems (AMAS) theory, **we identify a set of Non-Cooperative Situations (NCSs) susceptible to disturb the normal optimization process, and propose a set of cooperation mechanisms to handle them**.

We demonstrate the modularity of our system by **introducing additional concerns with the handling of uncertainties propagation**.

From this work follow two more general contributions concerning the design of MAS.

Using the *Make Agent Yourself* framework **we propose a component-based architecture for AMAS adapted to the handling of multiple agent roles and NCSs-related mechanisms**. This architecture is based on the idea of stackable skills components following the hierarchy of agent roles, providing the correct methods at the required level.

We also provide a more theoretical contribution **by abstracting the NCSs and solving mechanisms into more general *Collective Problem Solving Patterns* (CPSPs)**. These CPSPs are based on a more high-level agent role representation, and are abstracted from any direct application domain. They represent specific agents topologies which can be encountered in agents organizations leading to a disruption of the correct system function, as well as of solving mechanisms proposed to handle such configurations. We propose a schematic "blueprint" representation which synthesizes the content of the different patterns.

At last, **we integrate our MAS nto a working prototype and apply it on real-world problems** provided by our industrial partners Airbus and Snecma, in the context of the *Integrated Design for Complex Systems* project, a French National Research Agency funded project.

Manuscript Organization

This thesis is divided into 4 parts, divided in several chapters:

Part I. This part introduces the context of the study by presenting an overview of the continuous optimization field, MAS for optimization and the AMAS theory.

Part II. This part presents a MAS for solving continuous optimization problems. We propose a modeling of a continuous optimization problem as an agents graph, and describe the cooperative behaviors associated with the different agent roles.

Part III. The integration of our MAS into a working prototype is presented in this part. Based on our experience we propose a component-based architecture for AMAS agents and extend previous works on providing AMAS engineering tools by introducing CPSPs.

Part IV. In this part we present the experiments we did in order to evaluate and validate our approach.

An Adaptive Multi-Agent System for Self-Organizing Continuous Optimization

Context of the Study and State of the Art

1 Continuous Optimization

1.1 Basic Concepts

Before starting to present the different categories of optimization, we would like to take a moment to define what exactly *is* optimization.
In the most general way, optimizing is *trying to find the best element among an elements set*. When finding this best element is not trivial, we can rightfully talk about *solving an optimization problem*. This seemingly simple definition implies in fact quite a lot.

First of all it requires a defined set of elements to choose from. As we will see, the topology of the set is of the utmost importance when deciding of a method to solve the problem. This set of elements is often named the *search space*, *solution space* or *domain*. In "basic" optimization problems, the search space can be simply defined by a set of elements (for example $\{a, b, c\}$ or $\mathbb{R}$) associated with a set of *constraints*. For large problems, the search space can be defined by calculus-heavy equations, empirical models, complex algorithms... or even a mix of all of the above.

Search space - the set containing all the possible candidate solutions for the optimization problem.

While we said that the search space can be defined by a set of constraints, it is often more convenient to express the constraints separately. For example, if the search space of an optimization problem is defined over all the real numbers lower than 2, instead of defining the search space as $[-\infty, 2]$, it will usually be refereed as $\mathbb{R}$, with the added constraint $x < 2$. Usually we say the problem to be *subject to* (s.t.) the constraint $x < 2$.
In theory these two formulations should be equivalent. In practice however, these constraints are often the result of a real-world concern, and thus subject to some inherent imprecisions. Let us imagine an optimization problem defined by an engineer trying to design an aircraft. Among the constraints defined by the engineer, the aircraft weight must be lower or equal to 40t. Does it means that a solution for which the weight of the aircraft would be 40t *and 200g* would be completely unacceptable? Obviously not, the designer may be willing to accept some concessions in regard of the violation of this constraint. In engineering design, this is a common situation, and making these constraints explicit can be advantageous.

Since we want to find the best element of this solution space, we have to determine what makes an element better than another. Usually, the possible solutions are compared through a specific function called the *objective-function*, or *cost function*. The best element would be

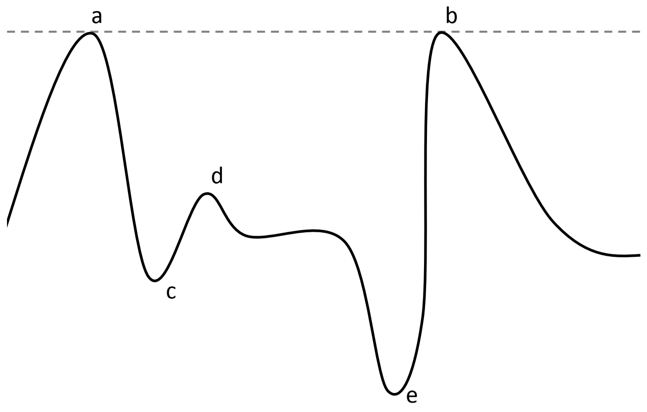


Figure 1.1: Examples of local and global optima - *a* and *b* are global maxima, *c* is a local minimum, *d* is a local maximum and *e* is the global minimum.

the one for which the objective function returns a minimal (or alternatively, maximal[1]) value. It should be noted that it is possible for a problem to admit several equivalent solutions in regard of the objective-function.

Objective-function - a function defined over the search space of the optimization problem, expressing the goal of a stakeholder for this problem.

When the search space is very large, or its topology is complicated, it can be really long or difficult to find the best solution and, more important, to be sure that the solution is the best. In fact, the only way to find the best solution with certainty may be an exhaustive exploration of the search space. Since it can be very costly regarding time and computation, instead of finding the best solution, we settle for the best *known*, which is considered to be "good enough", for example because this solution is the best of its neighborhood in a subset of the search space. The absolute best solution is called the *global optimum*, while the best solution in a neighborhood is called a *local optimum*. In a similar fashion, methods which try to find the global solution are said to be *global optimization methods*, while methods which are driven by local optima are said to be *local optimization methods*.

Optimizing - finding an element of the search space that minimizes (or maximizes) the value of the objective-function

On Figure 1.1, we can see different examples of global and local optima. The points labeled *a* and *b* are both global maxima, as they have the same value. The points *c* and *d* are respectively local minimum and maximum, while *e* is the global minimum.

From all the preceding, we can provide the minimal formulation of an optimization problem as follows:

[1]Obviously we sometimes want to find the *maximal* value that is solution of a problem, however minimizing $f(x)$ is equivalent to maximizing $-f(x)$. So maximization problems can be expressed as minimization problems, and *vice-versa*. Traditionally, optimization problems are usually expressed in the terms of finding a *minimal* value since the two possibilities are equivalents.

$$\text{minimize } f(x) \\ \text{for } x \in S$$

Where S is the search space of the problem and *f(x)* the objective-function.

1.1.1 Continuous versus Discrete Optimization

We must make an important distinction between *continuous* optimization (also called *numerical* optimization) and *combinatorial* optimization. The difference between the two categories concerns the definition domain of the variables, and consequently the nature of the search space. For combinatorial optimization, the variables can only take a finite number of different values (their definition domain is a finite and enumerable set), while the definition domain of continuous optimization problem variables is, as the name implies, continuous (thus the variable can take an infinite number of values).

This distinction has a profound impact on the nature of the problems. While the search space of a combinatorial optimization problem is finite, a continuous optimization problem can have an infinite number of potential solutions. Interestingly, this distinction does not implies that continuous optimization problems are inherently harder to solve than discrete ones. While combinatorial optimization problems can be NP-complete in the general case, some continuous optimization problems can easily be solved in polynomial time (even large ones). This surprising property comes from the fact that the complexity of combinatorial optimization problems comes from the difficulty to efficiently explore the different possibles variables states combinations, whereas for continuous optimization, the difficulty comes more from the shape of the search space. In simple continuous optimization problems, the search space will be very regular and the optimum easy to find, while optimization problems with complicated, “chaotic” search spaces will be much more difficult to solve.

For the sake of completeness, let us add that there exists an optimization field which is at the border between continuous and combinatorial optimization, named *Integer programming*[2], in which the variables must take integer values but are not restricted to a bounded definition domain (making their definition set countable but infinite). An extension of these problems are *mixed-integer programs*, where some of the variables are restricted to integer values and some are not. For more informations on integer programming, the reader can refers to [Sch98]. Integer programming and combinatorial optimization are sometimes regrouped under the *discrete optimization* category. However, the terms *combinatorial* and *discrete* seems to often be used somewhat interchangeably.

In the context of this thesis, we will concern ourselves more specifically with continuous optimization.

[2]in this context, “programming” must not be mistaken with computer programming but be understood as a synonym for “optimization”. This peculiar use of the word comes from the historical origins of the optimization field.

1.1.2 No Free Lunch Theorem

The No Free Lunch (NFL) Theorems[3] for optimization are important results in the field, formalized by Wolpert and Macready [WM97]. The basic idea behind these theorems is that no optimization method outperforms the others when regarding the class of all the possible problems or, as the authors themselves say:

> *"any two algorithms are equivalent when their performance is averaged across all possible problems."*

If an algorithm outperforms another on certain types of optimization problems, there must be other classes of problems where this algorithm performs worse.

The NFL theorems are based on several assumptions:

- ▷ the search space is finite,
- ▷ the optimization algorithms never re-evaluate the same point.

The first assumption limits the scope of NFL theorems to the realm of combinatorial optimization, as continuous optimization problems contain by nature an infinity of elements. Indeed, it has been shown that, in the context of continuous optimization, free lunches were possible [AT10] (but possibly at the cost of very big memory footprint). Thus this result does not impact directly the scope of this work, but we believe that it is still a good illustration of one of the main problematics of the optimization research field, which is that one must often compromise between universality and efficiency. Even in the context of continuous optimization, where the existence of free lunches has been demonstrated, it is probable that we will never find a be-all and end-all optimization technique. This point is for example discussed in [Yan12].

One example which can be connected to the NFL is the compromise between exploration and exploitation. Basically, an optimization method must often make a compromise between using the previous results to converge inside a region of the search space and exploring the rest of the search space to find a better region. For example: some gradient-based methods will use the evaluated points to converge toward a local optimum, but can miss a better solution as they insufficiently explored the search space. On the opposite, some other methods can make a thorough exploration of the search space (by partitioning or randomly selected points), but will be slow to converge toward the exact optimum. It is often possible to parametrize the method to tune the compromise between exploration and exploitation regarding the nature of the problem at hand, but a relevant parametrization requires a sufficient knowledge of the properties of the problem and therefore the parametrized method is only efficient for the specific problem.

Of course, the NFL theorems consider *all the possible problems*. One could argue that "interesting" problems (at least from an engineering point of view) are not distributed evenly over such a space, but correspond to a subset for which some optimization methods are more efficient than others. This is one of the reasons why optimization as a scientific research domain still makes sense.

[3]The term "no free lunch" comes from the popular English saying "there ain't no such thing as a free lunch", meaning that it is impossible to get something for nothing.

This distinction has been formalized by differentiating *incompressible* from *compressible* objective-functions. Incompressible objective-functions are random and it is thus impossible to develop an optimization method to find the solution efficiently, since good and bad values are randomly distributed. Of course, such "impossible" objective-functions make for the major part of all the possible search space [Eng00].
As we said, the set of "interesting" objective-functions, or even the set of real-world related ones, is much much more restrained. And for this specific category, some optimizers are better than others. However, as we will see in the next sections, the variety of "interesting" optimization problems is still important enough to have resulted in multiple specific optimization techniques.

One consequence of the NFL is that selecting an efficient optimization method for a given problem requires to have at least a minimum insight on the properties of the problem. No algorithm can be deemed to be the most efficient in the general case.

On a side note, it can be added that, in addition to the case of continuous optimization, the possible existence of free lunches (*i.e.* there exist optimization methods strictly better than some others) has been demonstrated for coevolutionary [WM05] as well as multi-objective [CK03] optimization problems.

1.2 Continuous Optimization Methods

As we have seen at the end of the last section, optimization methods have to make various compromises regarding applicability versus efficiency. A great number of methods exists in the literature, from general methods applicable to a variety of optimization problems to very specialized methods designed to be efficient in a specific context. These methods can be categorized by the type of problems they aim to solve or some inherent properties of the method.

Some possible criteria to discriminate based on the type of problem:

- can we obtain derivatives of the functions defined in the problem?
- is the problem linear (or possibly quadratic)?
- is the problem convex?

For the method in itself:

- is the method able to take constraints into account?
- does the method provide a global solution or a local one?
- is the method deterministic or stochastic?

Based on the shape of the search space and our requirements regarding the method, we must try to choose the most adequate optimization method. As always, the more information known regarding the problem, the more we will be able to select a specialized method with a great efficiency. If very few information is known, it could be necessary to use black-box optimization methods, that is methods that do not make any assumption regarding the nature

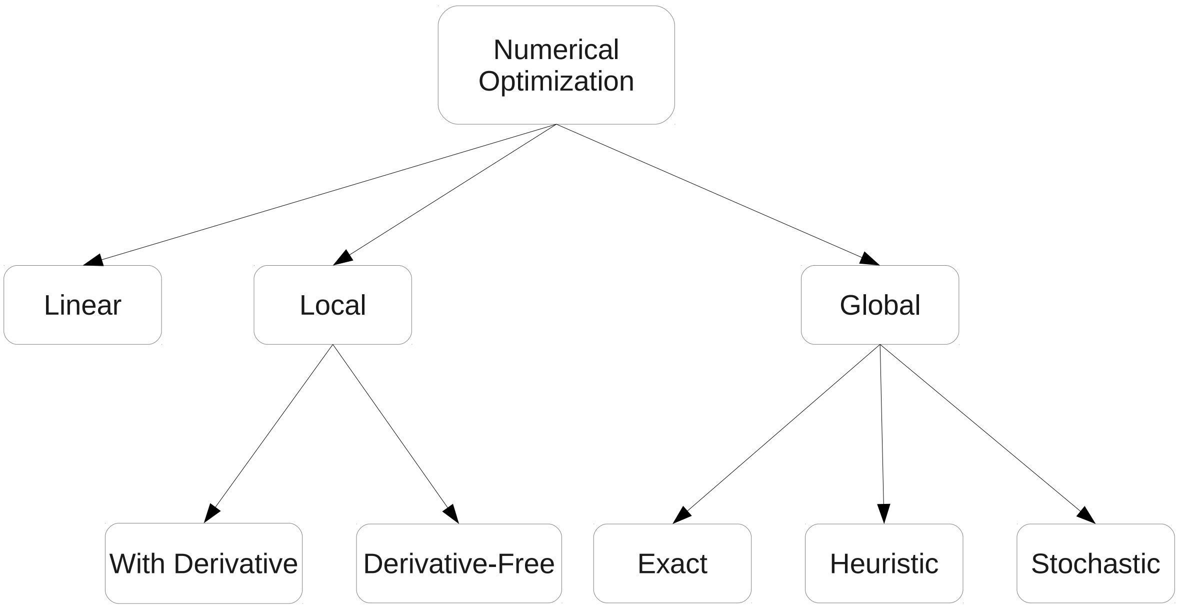


Figure 1.2: Types of continuous optimization methods.

of the problem. These techniques can suffice in themselves to get a good-enough result, or can be used as a prelude of more specialized techniques if enough information is gathered.

A rough proposal of organization of the different types of continuous optimization can be seen on Figure 1.2.

Presenting all the continuous optimization techniques would be far outside the scope of this work. In the next sections we will focus on presenting briefly the different optimization categories we identified on Figure 1.2. We will reference some of the most representatives techniques to illustrate this presentation, but without going too much into the details of the techniques.

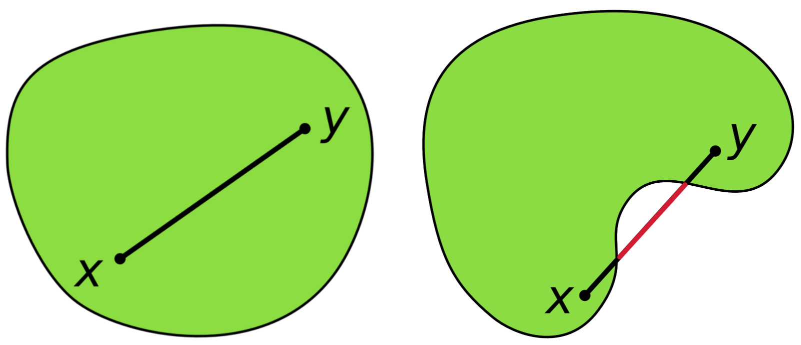
1.3 Linear Optimization

Linear optimization focuses on the solving of problems where the objective-function and constraints are linear (that is, $f(x+y) = f(x) + f(y)$ and $f(ax) = af(x)$), and the search space convex.

A *convex* search space is a set where all the lines segments linking the points of the set are fully contained inside the set (if it is not the case, the set is said to be *concave*). An illustration of the difference between convex and concave search spaces is shown on Figure 1.3.

A linear problem with n variables and m constraints can be expressed as:

$$\begin{aligned} \min\ & a_1x_1 + a_2x_2 + ... + a_nx_n \\ \text{subject to}\ & x_i \geq 0, \forall i \in 0, n \\ & a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n + b_1 \geq 0 \\ & ... \\ & a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n + b_m \geq 0 \end{aligned}$$



(a) Convex search space (b) Concave search space

Figure 1.3: Convex and concave search spaces (from Oleg Alexandrov).

This kind of problems is the most basic one. The most well-known method to solve linear problems is the Simplex algorithm [Dan98], published by Dantzig in 1947 (based on the work of Leonid Kantorovich). The Simplex algorithm is considered to be the first formal algorithm to solve a continuous optimization problem and to be one of the founding works of the field.

The basic idea of the simplex algorithm take advantage of the fact that the solution space of such problems is a convex polyhedron, for which the optimal solution is necessarily on one of its vertices. Another property is that a vertex which is not the optimal solution will have at least an edge leading toward a better vertex. The simplex algorithm starts on one of the vertices, and tries to follow an edge to a vertex which improves the objective-function. The algorithm iterates until it has found a vertex with no edge leading toward a better point (the optimal solution) or until it reaches an unbound edge (in this case the problem is not bounded).

While the simplex algorithm is considered to be very efficient for most cases, some alternatives have been proposed. The most noteworthy is the interior point method and its derivatives. Contrary to the simplex method, this method actually traverses the interior of the solution space (hence its name). The original method proposed by Karmarkar [Kar84] travels through the search space by iteratively finding the best solution over a restricted region delimited by a sphere around the current point.

1.4 Local Methods

Unconstrained local optimization methods concentrate on finding local optima either because the search space is too big or to complex to find a global optimum in a reasonable time, or because we prefer to quickly find a good enough solution. Moreover, if the search space is convex, the local optimum is also the global optimum. In such context local methods can be an easy mean to obtain the best solution.

1.4.1 Using Derivatives

If the derivatives are available, it is possible to propose very efficient optimization techniques to converge toward a local optimum.

In some cases, even if the derivatives are not available, it is possible to obtain an approximation of the derivatives, using for example a finite difference method. The basic idea of finite difference is to estimate the derivative from the ratio between the variation of the input and the variation of the output of the function. Basically, if we make the following assumption:

$$f(x+d) \approx f(x) + df'(x)$$

then we can estimate the derivative as follow:

$$f'(x) \approx \frac{f(x+d) - f(x)}{d}$$

1.4.1.1 Without Constraint

Methods of this kind can be mostly classified in two broad families: line search and trust region. Both families are iterative approaches but differ in the information they use to select the next search point.

The common idea of these methods is, from a random starting point x_0, to iteratively evaluate a new point using a direction and a step size. By adjusting the way the direction and step size are chosen, we can obtain a whole range of algorithms with varying behaviors.

Formally, from the point x_i, the new point x_{i+1} is calculated with:

$$x_{i+1} = x_i + s_i d_i$$

where d_i is the direction vector and s_i the step size coefficient.

Line search Line search is the most basic approach. It first determines which direction would improve the objective function, then decide of a step size to move toward it. Some of the most well-known linear search methods are gradient descent, Newton's and Quasi-Newton methods [DS83]. The gradient method simply uses a step size proportional to the value of the gradient. The Newton's and Quasi-Newton methods however try to find a solution to the equation $\nabla f(x) = 0$, where $\nabla f(x)$ is the gradient of f. The Newton's method uses the gradient and the Hessian matrix of f to estimate a good step size, while Quasi-Newton methods avoids the disadvantages of using the Hessian (which can involve costly operations), for example by replacing it with an approximation based on the variations of the gradient.

Trust region Trust region methods use the neighborhood of the current point to approximate the shape of the objective-function. This approximation is then used to find the minimum in a localized region around the current point. Contrary to line search methods, these methods start by selecting a step size (the size of the region around the current point) and choose the direction after. For example, the Levenberg-Marquardt algorithm [Lev44; Mar63], whose primary application is least squares problems, uses the Jacobian matrix conjugated with a damping factor to iteratively refine the approximate functions.

1.4.1.2 With Constraints

One of the first proposal for solving constrained problems with derivative is a generalization of interior point (presented in 1.3). The idea is that a linear function with a non-convex search space can be transformed into a linear function over a convex search space (which is a requirement for applying interior point techniques), using *self-concordant barrier functions*.

A barrier function is a function whose value increases to infinity when its input approaches a fixed boundary. The idea is to replace the constraint with a barrier function which is composed to the objective-function. The barrier function is thus used as a penalty for points that violate the constraint.

For example, taking the following constrained optimization problem:

$$\begin{aligned}&\text{minimize } f(x)\\&\text{subject to } x > 0\end{aligned}$$

Suppose we are provided with a barrier function $b_c(x)$ whose value increases toward infinity as x decreases toward 0 ($\lim_{x\to 0^+} b_c(x) = +\infty$). We can now use the new unconstrained optimization problem:

$$\text{minimize } f_{obj}(x) + b_c(x)$$

Another method, that dominated for some time this part of the continuous optimization field, is Sequential Quadratic Programming (SQP) [BT95]. SQP proposes to replace the problem to solve by a sequence of quadratic problems, usually more easily solvable. SQP is based on a very powerful result of continuous optimization, the Karush-Kuhn-Tucker (KKT) conditions [KT51]. The KKT conditions are necessary conditions for a solution $x*$ in a nonlinear optimization problem to be a local optimum. They state that for $x*$ to be a local optimum to an optimization problem with i inequality constraints gi and j equality constraints h_j, there must exist some μ_i and λ_j such as:

$$\begin{cases}\nabla f(x*) + \sum \mu_i \nabla g_i(x*) + \sum \lambda_j \nabla h_j(x*) = 0\\ g_i(x*) \leq 0 \ \forall i\\ h_j(x*) = 0 \ \forall j\\ \mu_i \geq 0 \ \forall i \quad \mu_i g_i(x*) = 0 \ \forall i\end{cases}$$

The SQP can be viewed as an equivalent of Newton's method to the KKT conditions of the problem.

1.4.2 Derivative-Free

Sometimes we cannot use derivatives in the optimization process, either because the derivatives are not available or too costly to compute.

A very popular method is the Nelder-Mead algorithm [NM65]. This algorithm places a simplex[4] on the search space and applies to it a sequence of transformations by moving its

[4]The concept of simplex is a generalization of the concept of triangle in arbitrary dimensions. A triangle

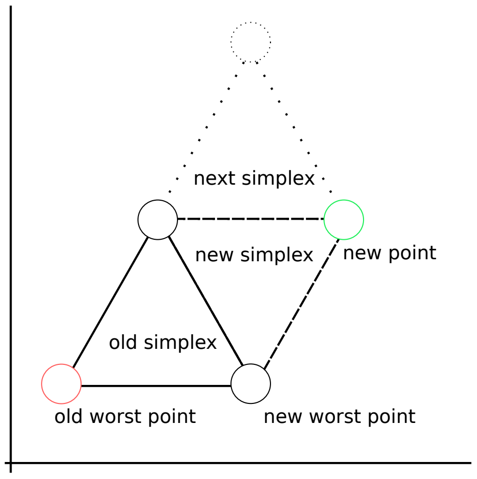


Figure 1.4: Illustration of the Nelder-Mead algorithm.

edges.

At each iteration, a new test point is calculated (for example the symmetric of the worst vertex of the simplex regarding the gravity center of the others points). If this point is better than every vertices, then the simplex is stretched toward it. If this point is worse than the current vertices, then we suppose we are stepping over the valley which contain an optimum. In this case the simplex is contracted to be able to explore the valley. Else, we simply replace the worst vertex by the new point. The iterations stop when the simplex has reached a specified size.
An illustration of the Nelder-Mead algorithm is shown on Figure 1.4.

Several derivative-free algorithms interpolate the objective-function with a simpler one. These methods starting with several evaluation points of the objective-function to build a simpler function on which it is possible to apply a known optimization method. Based on the result of the optimization, we can update the interpolation model and reiterate. Some possible interpolation methods include various polynomial interpolation methods, kriging *etc.*

1.5 Global Methods

While local optimization methods aim only for an optimum into a limited part of the search space, global optimization methods aim for the global best solution of the problem. These methods concentrate on providing strategies to explore the search space.

Depending on the nature of the problem, it may be impossible to guarantee that the best solution will be found by any means other than a complete exploration of the search space (which is in itself an impossible task in the context of continuous optimization).

For example, suppose the following problem:

is a simplex in 2 dimensions, a tetrahedron a simplex in 3 dimensions. It should be noted that the Simplex algorithm, presented in 1.3 does not actually use simplices during solving, despite its misleading name, but is simply inspired from the concept.

$$\text{Minimize } f(x) = \begin{cases} 0 & \text{if } x = 1 \times 10^{-9} \\ 1 & \text{otherwise} \end{cases}$$

There is no efficient strategy to find the global optimum to such a problem (unless knowing the equations, and thus the solution, beforehand).

Consequently, obtaining the global optimum of an optimization problem with certainty will be possible or not depending on the properties of the problem. As with local optimization methods, several kinds of approaches have been proposed to solve problems with different properties.

1.5.1 Exact Methods

These methods aims at providing a guarantee about the optimality of the solution. They cannot be applied to every optimization problems, as they need to use some properties of the problem to prove that the solution found is a global optimum.

For example, as we said before, if an optimization problem is convex then a local solution is also a global solution. Consequently, for convex problems local optimization methods can be used with the guarantee that the solution found will be the best one.

As with linear programming, a specific class of problems belongs to the field of *Quadratic Programming* (QP), and its extension *Quadratic Constrained Quadratic Programming* (QCQP). QP concerns (non-convex[5]) problems where the objective-function is quadratic (that is, a polynomial of degree 2) and all the constraints are linear. QCQP, as its name implies, concerns quadratic objectives and constraints.

Several adaptations of the Simplex method (presented is 1.3) for QP have been proposed (see for example [Wol59; Dan98; PW64]).

It is also very common to use relaxation techniques (*Reformulation-Linearizion Techniques* or *Semidefinite Programming*) to approximate the quadratic problem with a linear one. A widespread technique is to use a Branch and Cut, starting by replacing each non-linear term of the problem by a (linear) variable. For example, the term $x_i x_j$ would be replaced by the variable v_{ij}. For each replacement, a new constraint is added to maintain the consistency of the solution. In our example, the new constraint would be $v_{ij} = x_i x_j$. Then these new constraints are themselves replaced by linear approximations. To see more details on the different relaxations strategies, refer to [Aud+00].

1.5.2 Heuristic Methods

When exact methods are too costly or non-applicable, it is often sufficient to get a good enough solution. To this end it is possible to use heuristics to reduce the search space and ensure we obtain a solution in a reasonable time.

For example, a simple strategy can be to run a local optimization technique several times with different initialization points, in order to increase the coverage of the search space.

Expanding on this strategy is the *Tabu search* metaheuristic [Glo89]. The idea of Tabu search is to execute the same algorithm several times. To avoid converging toward the same

[5]if the problem is convex, then we can use simple convex optimization techniques

point, the previous solutions are memorized and are not allowed to be revisited. Thus the method has less chances to be quickly trapped in local optima. The Tabu search can be extended with several levels of memory to influence the search toward the most interesting regions of the search space.

Another class of heuristics introduces randomization into the search process in order to obtain a good coverage of the search space. These strategies are said to be *stochastic*.

A very simple example of stochastic method is the Monte-Carlo method [RC04]. We start by choosing a point at random. Then we iteratively draw new points around the old one until we cannot improve the solution. Then we start again from another random point of the search space and continue until we reach an arbitrary termination criterion. This method is very easy to put in practice, but can be quite costly in number of evaluations.

Another well-known technique is Simulated Annealing [KJV83]. This method is inspired from the annealing technique in metallurgy used to control the heating and cooling of a material in order to increase its properties. Simulated annealing associate a global temperature and an energy measure to the state of the system (which represent the objective-function). Consequently the goal of the system is to minimize its energy. At each step, the method considers moving to one of the neighbors of the current point. The higher the temperature of the system, the more the system is susceptible to move, even if the neighbor is worse than the current point. As the temperature decreases, the system will favor more and more consistently solutions that decrease its energy. The temperature of the system is reduced over time following a specific cooling schedule. For example the temperature could be decreased a each step, or could be reduced only when the system reached an equilibrium.

Some others stochastic techniques which can be applied are population-based optimization. Population-based optimization is a very prolific and diversified field, whose algorithms are often inspired from the observation of natural populations behavior. The underlying principle of this kind of algorithm is the use of a population of entities which are spread within the search space, with specific strategies in order to iteratively progress toward the solution. These strategies involve transformations applied to the entities that compose the population (moving, creating or destroying or otherwise altering them).

1.5.2.1 Evolutionary Algorithms

Evolutionary Algorithms (EA) are based on a Darwinist selection process. The main idea is to maintain a population of solutions which is iteratively improved by selecting, crossing and mutating the best individuals in regard of a given *fitness function*. Most EA are based on the algorithm shown in algorithm 1.1.

Genetic algorithms [Hol92] are by far the most often used type of EA in optimization. In the context of genetic algorithms, each individual entity is defined by a specific *genotype*, which represents a possible solution to the problem. Like we said previously, a global *fitness function* is defined, which is used to evaluate the solutions represented by these individuals.

Regarding a continuous optimization problem, the genotype of an entity would be a set of values assigned to the inputs of the problem, while the fitness function would be the objective-function.

The optimization process iterates over two phases: *selection* and *generation of a new*

Algorithm 1.1: Evolutionary Algorithm Pseudocode

```
generate initial population
repeat
    evaluate fitness and select best-fit individuals
    produce offspring based on selected individuals
    evaluate fitness of offspring
    replace least-fit population with offspring
until termination condition reached
```

population. In the selection phase, a subset of the current population is selected to be used in the generation of the new population. In the most basic approaches, the selected individuals are the ones which obtain the best scores on the fitness function. The second phase generates a new population, based on the selected individuals. To produce the new individuals, a set of biology-inspired genetic operators is used. Two of the most common genetic operators are *crossover* and *mutation*. The crossover is the creation of a new individual, whose genotype is based on the combination of two randomly selected "parents". The mutation consists in randomly changing a part of the genotype of an individual.

Together, these two operators can be seen as representatives of the exploitation versus exploration dilemma. The crossover operation is an attempt to exploit the best solutions found so far, while the mutation is done to maintain diversity in the population.

Most genetic algorithms proposals work on variations on these two phases. For example, some techniques will select not only the best fitted individuals, but also a small part of low fitted ones in order to increase the exploration of the search space. Some variants also modify the genetic operators or introduce additional new ones. For example it is possible to do the crossover using more than two parents, or to introduce the notions of groups and migration.

1.5.2.2 Swarm Algorithms

Swarm algorithms focus on the decentralized exploration of the search space by a population of entities (commonly called *agents* or *boids*). These entities usually have only a local perception of the search space and individually decide how to move using simple rules.

Particle Swarm Optimization (PSO) [KE95] is inspired from the flocking behavior of social animals (such as birds, fishes or bees). PSO consists of a swarm of particles initially spread across the search space. Each particle i can perceive its neighborhood and moves according to its current position x_i and velocity vector $\overrightarrow{v_i}$. During solving, each particle memorizes the best solution it found so far x_i^*. Based on this information, a global best x_g^* is calculated and provided for each particle for the next iteration. At each step, a particle updates its velocity based on the current velocity, its local best value and the global best value found. Then it moves according to its current position and velocity. The influence of these various parameters can be modulated using the coefficients w, representing the inertia, c_l representing the influence of the local best solution and c_g, representing the influence of the global best solution.

Algorithm 1.2 presents the principle of an iteration of PSO.

Algorithm 1.2: PSO iteration

```
foreach particle do
    $r_l, r_g \leftarrow$ random numbers $\in [0,1]$
    $\overrightarrow{v_i}(t) = w\,\overrightarrow{v_i}(t-1) + c_l r_l(x_i^*(t-1) - x_i(t-1)) + c_g r_g(x_g^*(t-1) - x_i(t-1))$
    $x_i(t) = x_i(t-1) + \overrightarrow{v_i}(t)$
    if $x_i(t)$ better than $x_i^*$ then
        $x_i^* = x_i(t)$
    end
end
$x_g^* = \max(x_i^*)$
```

1.5.2.3 Population-Based Algorithms for Optimization in Dynamic Environments

A special case of optimization is optimization in a dynamic environment. In this kind of problems, the objective-function is likely to change with time. To avoid confusion, optimization in dynamic environment is distinct from *dynamic programming* (another optimization field not addressed here) in the fact that the changes of the objective-function *are not known beforehand*. It is thus not possible to anticipate these changes during solving. Many optimization techniques fail to solve this kind of problems as they are not meant to take into account these dynamics. These problems require optimization techniques to be able to both find a moving optimum and to follow it when it changes.

Population-based algorithms tackle the problem of a dynamic environment by maintaining a diverse population of solutions. Of course this kind of algorithm does not avoid the dilemma of exploration versus exploitation, and must usually choose between maintaining a high diversity of solutions and concentrating the population around the most promising regions. Most of these techniques include specific parameters to adjust the solving process toward more exploration or more exploitation (for example the coefficients regarding current velocity, local and global optima in PSO). The choice of finding the best parameters is once again handed down to the person applying the method.

Another inconvenient of population-based approaches is the high number of evaluations they require. Indeed, each individual will need at least one evaluation of the whole problem. If the problem is very costly to evaluate, this kind of method may not be worth considering.

Finally, this kind of method often has some difficulties to scale with the complexity of the problem. Genetic algorithms will often have difficulties solving problems where the search space is very large, and swarm algorithms will present difficulties when the search space has a high number of dimensions or with chaotic topologies.

1.6 Analysis of Continuous Optimization

This overview of continuous optimization illustrates how the different optimization techniques range from very efficient but highly specialized methods to broad scope methods with slower and more costly strategies. We have seen how the intuition provided by the NFL theorems is illustrated by the progression of the optimization techniques in general applicability and complexity.

An interesting observation is how specialized methods can concentrate on exploitation of the solutions, while generic methods are concerned with exploration of the search space. Indeed, the more we know about the structure of the problem, the easier it is to select a new search point. When lacking knowledge about the topology of the system, we need to be more concerned about exploring the search space.

From this basic compromise we can see how continuous optimization has given birth to such a plethora of methods, as every optimization technique is torn between this basic compromise.

All the methods we presented in this chapter share an inherent limitation due to the way they handle the problem. By using a centralized algorithm, with a global view of the problem, in charge of the evaluating each point and deciding which new point to explore, these approaches require a complete re-evaluation of the entire problem at every iteration. In the next parts we will see increasingly complex problems, where the size of the problem, its topology and additional concerns not strictly related to the pure optimization make a complete evaluation of the problem increasingly costly, up to a point where applying methods such as the one we have presented in this chapter becomes too prohibitive.

2 Multi-Objective Optimization

Multi-Objective Optimization (MOO), also called multi-criteria optimization, departs significantly from previous categories of optimization in the fact that you have to consider multiple objective functions instead of only one. A main aspect of MOO is the way to conciliate these objectives, that are usually contradictory. An example of real-world everyday MOO problem could be choosing the mean of transportation for a travel, trying to find a balance between speed and cost. Airplane is the fastest way of transportation, but is expensive. While car is slower, it is cheaper (especially if several people share the car). Train is slower than plane, more expensive than car, but can preferred as the best compromise. However, some solutions are strictly worse than others (in our example, renting an helicopter would probably be both more expensive and slower than buying a seat on a commercial airplane).

From this example we can see that, for an MOO problem, there is rarely a clear-cut "best" solution. And, more importantly, that even some solutions which are not optimum for *any* of the objectives can be deemed satisfying compromises. Only when each objective is completely independent, or when no objective is contradictory to another, then an MOO problem can be handled as a set of separated mono-objective optimization problems. A solution vector which would be optimum for each objective is sometimes called *utopia point*, or *shadow minimum*, and is used as a reference comparison in some of the MOO techniques we present.

The formulation of a MOO problem does not differ significantly from the formulation of a "classical" optimization problem, the only difference being that instead of minimizing one objective function, we aim to minimize several ones:

$$\begin{aligned} \text{minimize } & f_1(x), f_2(x), ..., f_n(x) \\ \text{for } & x \in S \end{aligned}$$

MOO problems are quite a radical departure from previous optimization problems we have seen. Many approaches have been proposed, the majority of which can be separated in two categories: a priori and a posteriori approaches. **A priori approaches** aim at discriminating between the objectives *before* the optimization process. This often consists in combining the different objectives into a single one, before applying a classical optimization method on the new, aggregated objective.
On the opposite, **a posteriori methods** try to provide a set of efficient solutions among which the decider will choose. A priori approaches are considered easier, but not very efficient,

whereas a posteriori approaches provide more diversity of solutions as well as more insight about the nature of the problem.
A third category can also be considered: the *interactive* methods. Basically, these methods iterate between decision and search phases. For example, an interactive method could work by quickly providing intermediate solutions to the decision-maker, which would in return refines the search using them.

We will now see some of the strategies that have been proposed to deal with such problems. A recommended reading for the interested reader is the very complete overview by Marler *et al.* in [MA04].

2.1 A Priori Methods

A priori methods usually propose a strategy to make possible the use of classical single-objective optimization methods on the problem. This may be done by discriminating between the different objectives or by aggregating them into a new objective which will be used for the optimization.

2.1.1 Objectives Aggregation

The first approach is to transform the MOO problem back to a mono-objective optimization problem, by aggregating the different objectives into one. This can be expressed as follow: $f_g = aggr(f_1, f_2, ..., f_n)$, where $f_1, f_2, ..., f_n$ are the original objectives and f_g the aggregated one, which will be used with classical mono-objective optimization methods.

Concerning the choice of the aggregation function, different strategies can be used. The simplest strategy is to use a classical function such as addition, multiplication, mean, max or min of the objectives, and variations of the preceding (exponential sum, ...). These methods present the major drawback of requiring the aggregated values to be comparable. Going back to our travel example, is it relevant to simply add together duration and cost?

A slightly more sophisticated way is the weighted sum method, where a coefficient is attributed to each objective before adding them [MA10] :

$$f(x) = \sum_{i=1}^{n} w_i f_i(x)$$

where w_i are the coefficients representing the relative preferences between the objectives.

This method allows to express a preference between different objectives, as well as bringing different objectives to a comparable scale. However, one now has to decide the values to select for the coefficients.
Furthermore, this method can hide some information concerning the solution, for example an extremely poor result in one of the objectives, compensated by small improvements in all the others. The method also presents some limitations in the fact that it does not guarantee that the final solution will be an acceptable point, nor that a continuous variations of the points will leads to a continuous distribution of the solutions. It is also not possible to obtain solution points situated in a non convex region of the solution set [MA04].

2.1.2 Lexicographic Method

In this iterative method, the objective-functions are arranged and optimized by order of importance. The result of the optimization at a given step becomes a constraint to satisfy for the following steps.

Formally, the problem becomes an ordered set of optimization problems, where each iteration can be expressed as follow:

$$\begin{aligned} &\min f_i(x) \\ &\text{subject to } f_j(x) \leq f_j(x_j^*) \\ &\text{for } j = 1, ..., i-1 \end{aligned}$$

It is also possible to replace inequalities by equality constraints [Sta88]. Another variation, sometimes called hierarchical method [Osy84], introduces a constraint relaxation coefficient δ where the new formulation of the constraints becomes:

$$f_j(x) \leq \left(1 + \frac{\delta_j}{100}\right) f_j(x_j^*)$$

2.1.3 ϵ-constraint Method

The ϵ-constraint method [HLW71] proposes to change the expression of the problem by keeping only one objective (the one deemed the most important) and transforming the others into inequality constraints.

For example, the problem

$$\min f_1, f_2, ..., f_n$$

where f_1 is deemed the most important objective would be transformed in

$$\begin{aligned} &\min f_1 \\ &\text{subject to } f_2 \leq \epsilon_2, ..., f_n \leq \epsilon_n \end{aligned}$$

where $\epsilon_2, ..., \epsilon_n$ must be chosen by the designer. Depending of the selection of $\epsilon_2, ..., \epsilon_n$, there is a risk to obtain a formulation where no feasible solution exists. See [MA04] for a discussion about the different proposed methods for selecting ϵ.

2.1.4 Goal Programming

The idea of Goal Programming (also called Target Vector Optimization) [CC57] is to assign to each objective an associated goal to reach. Each objective can be over- or underachieving its goal, allowing the designer to provide a rough idea of the initial design goals.

The new objective is to minimize the total deviation $\sum_{i=1}^{n} |f_i - g_i|$ where g_i is the goal associated to objective f_i.

Goal Programming is very easy to use and quite popular, and can work even when the designer provides some unreachable goals, but give no guarantee about the optimality of the solution.

2.1.5 MinMax Method

This method uses the separately attainable minima of the objectives (the so-called *utopia point*) and try to minimize the maximum deviation of the objectives relative to these minima [Osy84]:

$$\min \ \max \left(\frac{f_i - f_i^0}{f_i^0} \right)$$

where f_i^0 represents the separately attainable minimum of the objective f_i.

This method can also be used similarly to Goal Programming, where the separately attainable minima are replaced by goals given by the designer.

A variant of this method, called weighted min-max, or weighted Tchebycheff method, uses the following formulation:

$$\min f = \ \max w_i |f_i - f_i^0|$$

where w_i are coefficient provided by the designer.

2.1.6 Analysis of a priori methods

A priori methods provide a simple and efficient way to tackle the problem of multiple objectives, as they allow to reduce the problem to a mono-objective one. However, a drawback of these methods is the need for the designer to have a good knowledge of the problem, to know the correct way to combine/compare the different objectives functions. In the general case, the designer may not have enough experience or information to make such decisions.

In the case where the designer would have enough knowledge to meaningfully use such methods, the introduction of this knowledge may introduce a bias in the resulting solution, orienting the optimization process toward "standard" solutions at the expense of possible non-conventional ones.

Moreover, aggregation techniques tend to break when the objectives are not comparable, requiring once more knowledge from the designer to introduce adjustment coefficients in order to "re-equilibrate" the aggregation function.

2.2 A Posteriori Methods

We have seen that, while convenient, *a priori* methods can be quite restrictive. By choosing beforehand a way to aggregate the different objectives, we lose in diversity of solutions and influence the result of the optimization process.

2.2.1 Pareto Dominance

A radically different approach has been proposed, using the concepts of Pareto dominance and Pareto optimality. These concepts were originally developed in economical sciences, first by Francis Edgeworth and later Vilfredo Pareto [PPS72]. The initial application of the concepts was to propose a minimal definition of "efficiency", regarding allocation of resources inside an economical system.

The main idea is define as *Pareto efficient* (*Pareto optimal*) a state where it is impossible to improve the resources allocation for an individual without worsening the situation of at least another[1].
Conversely, if from a system state A it is possible to find a new state B where at least one individual's situation is improved without worsening the situation of another, the state A will be said to be *Pareto inefficient*. The state B will be said to *dominate* the state A in terms of Pareto optimality, and the passage from A to B will said to be a *Pareto improvement*. This relation of Pareto dominance is usually noted $\prec$.

Pareto-dominance - Given A and B two vectors describing resources allocations to different individuals in a system, $A \prec B \Leftrightarrow (\forall i\ A_i \leq B_i \wedge \exists j\ A_j < B_j)$.

Note that, to complicate a little the understanding of this notion, the definition of domination depends of whether we actually want to *maximize* or *minimize* the resource allocation. In the previous economical definition we wanted to maximize resources allocation, so $A \prec B$ reads "B dominates A". If we want to *minimize* the allocation, the meaning is inverted and $A \prec B$ reads "A dominates B".

Based on this relation of dominance, it is possible to provide a definition of Pareto-optimality.

Pareto-optimality - A solution vector that is not dominated by any other possible solution is said to be Pareto-optimal.

Pareto front - the set of Pareto-optimal solutions.

It is also possible to classify the solutions by ranks: a solution which is dominated by no other is said to be of rank 0 (and to be Pareto-optimal). A solution which is dominated by a solution of at most rank 0 is said to be of rank 1 and so on.

These definitions of dominance and optimality can be used to characterize the possible solutions of MOO problem. In this case, the problem is not to find an optimal solution anymore, but to find the Pareto front of the problem.

An illustration of a Pareto front can be seen on figure 2.1. The elements A and B are Pareto-optimal, while C is not, being dominated by A and B.

[1]As a remark, these definitions of efficiency and optimality do not give any information about the fairness of the allocation, or the well-being of the involved parties. From this point of view, a monopolistic situation where one individual would control all the available resources is as optimal as a situation where all the resources are equally divided between the individuals.

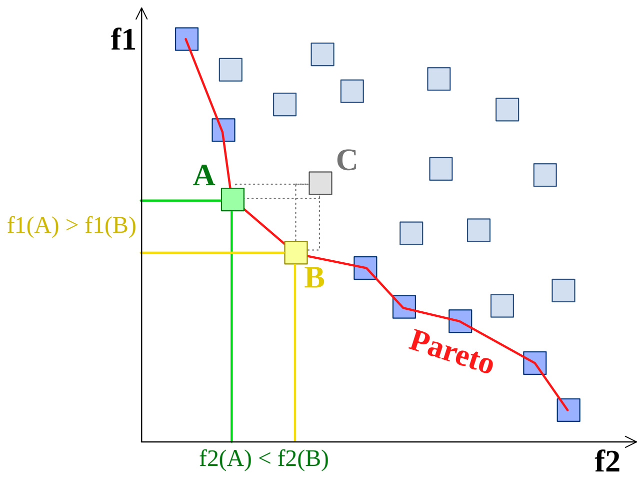


Figure 2.1: Illustration of the notion of Pareto Front (CC-BY-SA Johann Dréo).

2.2.2 Multi-Objective Evolutionary Algorithms

A popular strategy to find the Pareto front is to use specialized variants of population-based heuristics methods (presented in section 1.5.2). One of the most widespread approaches concerns Multi-Objective Evolutionary Algorithms (MOEA).

MOEA are variants of EA specialized in the simultaneous tracking of several solutions, instead of providing a single solution. Historically, MOEA are divided in two main categories, based on whether or not they use elitism mechanisms, with some consensus concerning the superiority of elitist algorithms.

Non-elitist techniques often have trouble to converge toward the Pareto front, as well as to keep a good optimum diversity and to spread on the front. To remedy that issue, elitist algorithms propose various additional mechanisms:

- maintaining an external "archive" population containing the optimum found so far
- using clustering techniques to spread the solutions among the Pareto front
- introducing an additional preferential bias toward non-dominated solutions

MOEA propose an interesting Nature-inspired mechanism for solving MOO problems. By maintaining a population of candidate solutions and using selection pressure mechanisms, they make possible a throughout exploration of the Pareto front.
The compromise between exploration and exploitation will be determined by the parametrization of the different genetic operators and the selection/crossing process. While this aspect can prove to be advantageous for the potential flexibility it brings, it is often difficult to guess what are the relevant parameters values. Consequently it can be necessary to iterate the optimization process in order to manually fine-tune the parameters.

A major limitation of these methods is their potential computational cost. Such algorithms need to evaluate a relatively large number of candidates in order to create a good population of solutions. Computing all the candidate solutions can be prohibitive for computationally expensive problems.

It has also be noted in [CK07] that many MOEA have poor scalability performances in regard of the number of objectives, often having troubles to handle more than five objectives.

This degradation of performances is attributed to the combined factors of:

- ▷ the exponential complexity increase of the procedures in regard of the number of objectives
- ▷ the increase in number of non-dominated solutions caused by the additional objectives
- ▷ the limited size of the archived population in regard to the increasing number of candidates

2.3 Analysis of MOO

We have seen how a new concern, taking in account several objectives, has created the need for new optimization strategies. Interestingly, many of these MOO-specific methods do not concentrate on the optimization process in itself, but on providing strategies to apply classical optimization techniques.

Some of these methods propose to aggregate the objectives, in order to be able to directly apply mono-objective optimization techniques. However these methods require the expert to make some choices concerning the priorities of the objectives before having the possibilities to look at the different alternatives.
Others approaches use population-based algorithms in order to maintain a set of solutions. However these approaches, already quite costly in the mono-objective case, require even more evaluations of the problem.

We will see in the next chapter how these methods are still insufficient to handle the most complex continuous optimization problems, due to the very fundamental assumption they make by considering the objective-function(s) to be trivial to evaluate. We will now see a kind of optimization problems which are so complex that a single evaluation of the problem is considered a cost, and some methods which have been proposed to handle such problems.

3 Multidisciplinary Optimization

Multidisciplinary Design Optimization (MDO), often abbreviated Multidisciplinary Optimization, concerns the optimization of complex systems which involves several interacting disciplines. Each discipline in itself can contain this own variables, objectives and constraints. These problems often involve several of the optimization problematics we examined in the previous chapters (non-linearity, multiples objectives and constraints, uncertainties *etc.*) and are usually too complex to be handled by classical optimization methods for several reasons. Evaluating the global function of the problem is considered to be expensive, as it involve complicated models, requiring extensive calculus. The optimization problem usually contains not just contradictory objectives, but whole conflicting disciplines. The complexity of the problem is also increased by the fact that the different parts of the problem are interdependent and can potentially add several layers of intermediate calculus, making difficult to estimate the impact of the design variables on the different criteria.
In this regard, we can say that MDO problems regroup and amplify all the difficulties encountered with the previous types of optimization problems.

These kinds of problems are commonplace in the industry, especially in complex systems design such as aeronautics and aerospace engineering, where parts of the design are often done by different experts teams. For example, designing an aircraft can be formalized as a MDO problem involving several disciplines such as mechanics, aerodynamics, acoustics etc. (see Figure 3.1).

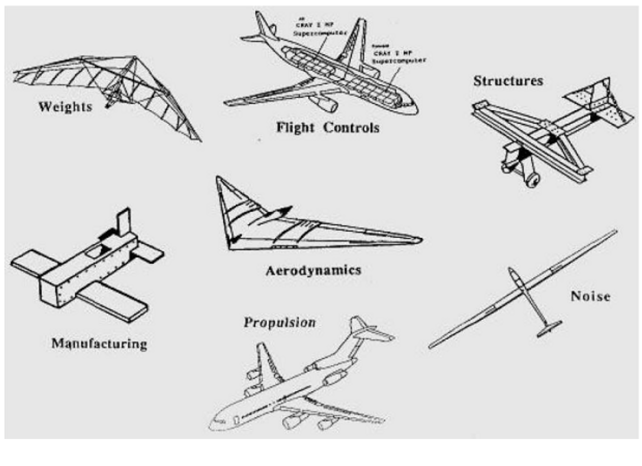


Figure 3.1: Examples of aeronautics disciplines (source unknown, partially based on the work of C. W. Miller *Dream Airplanes*).

To handle such complex systems, most strategies propose to decompose the problem into several sub-systems of lesser complexity. Concerning engineering, several decomposition strategies have been proposed[KL95]:

- ▷ Product (also called object) decomposition, based on the physical components of the system. This kind of decomposition is not always adequate and often subjective.
- ▷ Process (also called sequential) decomposition, based on the workflow of elements/informations involved into the design process. This decomposition is most adequate when the design process is linear.
- ▷ Domain (also called aspect) decomposition, based on the knowledge domains, the disciplines, involved. This kind of decomposition is the basis of MDO methods.

The AIAA MDO Technical Committee proposed the following definition of MDO[1] [AA91]:

"A methodology for the design of complex engineering systems and subsystems that coherently exploits the synergism of mutually interacting phenomena."

MDO methods are not optimization methods *per se*. Instead they focus on providing an optimization strategy for optimizing the different disciplines while maintaining a global coherence. In fact, the optimization of the disciplines is done using classical optimization methods such as the ones presented before. In this regard, MDO methods could be seen as optimization meta-methods, or methodologies, as they provide methods to best apply optimization methods to complex problems. Martin and Lambe [ML12] note many of the terms that have been used in the literature: "architecture", "method", "methodology", "problem formulation", "strategy", "procedure" or "algorithm".

Alexandrov and Lewis illustrated their discussion on Collaborative Optimization [NMRM00] with the following theoretical test case:

$$\begin{aligned} a_1 &= A_1(s, l_1, a_2) \\ a_2 &= A_2(s, l_2, a_1) \\ \text{minimize } & f(s, a_1, a_2) \\ \text{subject to } & g1(s, l_1, a_1) \geq 0 \\ & g2(s, l_2, a_2) \geq 0 \end{aligned}$$

It can be noted that this formulation does not differ from the standard optimization problem formulation. Indeed, as noted by Martin and Lambe [ML12]:

"If we ignore the discipline boundaries, an MDO problem is nothing more than a standard constrained nonlinear programming problem: we must find the values of the design variables that maximize or minimize a particular objective function, subject to the constraints."

[1] `https://info.aiaa.org/tac/adsg/MDOTC/Web%20Pages/aboutmdo.aspx`

A very common strategy used by most MDO methods is to reformulate the problem to *decouple* variables which are shared among the disciplines. For example, the following optimization problem:

$$\text{Minimize } f(f_1(x), f_2(x))$$

(where f_1 and f_2 represent two disciplines depending on x)

could become:

$$\begin{aligned}&\text{Minimize } f(f_1(x_1), f_2(x_2))\\&\text{subject to } x_1 = x_2\end{aligned}$$

The shared variable x has been replaced by two independent variables x_1 and x_2, and a new constraint $x_1 = x_2$ has been added to ensure the consistency of the design.

Several specific terms are in use in the domain of MDO:

- *Design variable*: a variable of the problem which can be chosen by the designer. The goal of the optimization process is to find good values for the design variables of the problem. A design variable is said to be *local* (or *private*) if it is relevant to only one discipline, and *shared* (or *public*) if it is used by several of them.
- *Discipline Analysis*: The evaluation of the output variables of a single discipline, based on given values for the input variables, ensuring that all the values involved in the discipline are consistent.
- *Multidisciplinary Analysis (MDA)*: The evaluation of the complete problem based on given values for the input variables. In order to find a set of consistent values, the different disciplines may need to be evaluated several times.
- *Optimizer/Solver*: A classical optimization technique, such as the ones we have seen in the previous chapters. These optimizers can be applied to the problem as a whole or to specific parts.

The classical approach to categorize MDO methods is to separate mono- and multi-level methods.
Mono-level methods use a single optimizer and a non hierarchical structure, while multi-level methods use a hierarchical structure and possibly several optimizers.

3.1 Mono-Level Methods

3.1.1 Multidisciplinary Feasible

MultiDisciplinary Feasible (MDF) [Cra+94], represented in Figure 3.2, is the most basic and classical MDO method. This approach ensures at each optimization step that the design

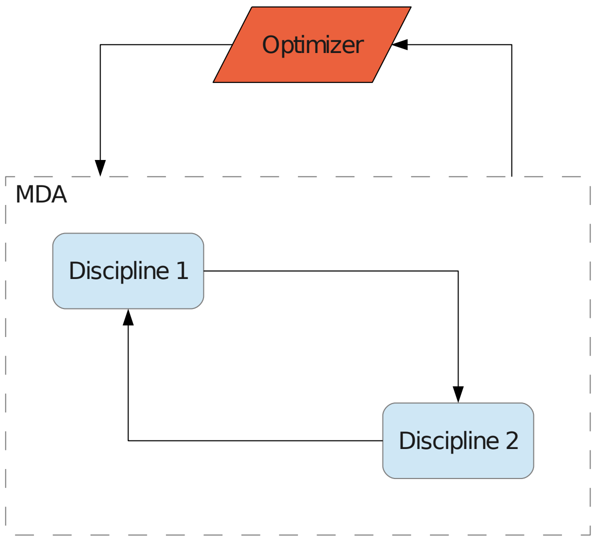


Figure 3.2: MDF method.

is consistent as a whole, taking into account all the disciplines together (hence the name). The optimizer only uses the design variables, objective-functions and constraints. Basically, MDF alternates between a MDA (multidisciplinary analysis) and a global optimization phase, ensuring the design to be globally consistent at each optimization step. At each step, the result proposed by the optimizer is used to do the full MDA whose results are used in return for the next optimization iteration.

As it is so straightforward, MDF requires no reformulation of the problem, unlike most of the other MDO methods, and in so is really easy to use. As the design is consistent at each step, the optimization process can provide a solution at any time (but it will not guarantee that the proposed solution will satisfy the constraints, as this concern depend on the optimization technique used). However, since MDA is supposed to be costly, MDF is often considered to be quite inefficient, since it never exploits the parallelization opportunities due to the separation of the disciplines. MDF does not provide any guarantee of convergence.

3.1.2 Individual Discipline Feasible

Individual Discipline Feasible (IDF) [Cra+94], represented on Figure 3.3, differs from MDF in the way that it ensures at each step consistency for each discipline separately, but not consistency between disciplines. The global consistency of the system is not ensured until convergence. Instead of a full MDA (as in MDF), IDF alternates the optimization with independent disciplines analysis.

However, as the variables shared among the disciplines are not guaranteed to be consistent, IDF needs to introduce a reformulation of the problem where the shared variables are duplicated among the disciplines, and several equality constraints are added to ensure the eventual consistency.

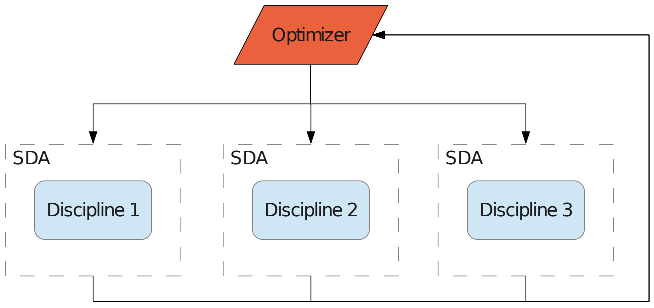


Figure 3.3: IDF method.

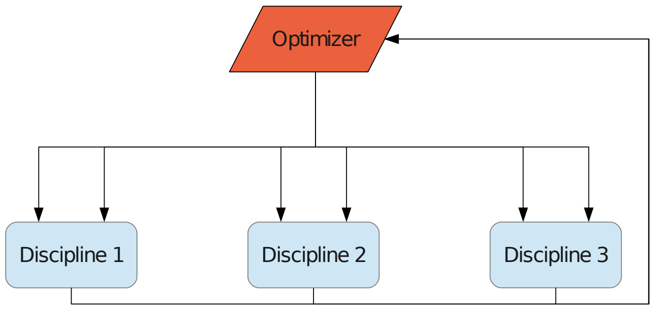


Figure 3.4: AAO method.

3.1.3 All-At-Once

All-At-Once (AAO) [Haf85; Cra+94], represented on Figure 3.4, can be seen as the extreme opposite of MDF, given that it does not try to maintain consistency neither at the global nor discipline level during the optimization process until it reaches convergence. All variables are considered as design variable for the optimizer, and the analysis equations are transformed into equality constraints.

This transformation allows the analysis phase to be very quick, as we only need to evaluate the residuals of the equality constraints representing the equations. However, the drawbacks already present in IDF are even more important, as AAO requires an even bigger reformulation of the problem, introducing many duplicated variables and consistency constraints to the problem. This reformulation also makes the optimization phase more complex.

3.2 Multi-Level Methods

3.2.1 Concurrent Subspace Optimization

Concurrent Subspace Optimization (CSSO) [WRB97], which can be seen on Figure 3.5, is one of the first multi-level MDO methods. Before the optimization, the problem is decomposed in several subspaces related to the different disciplines. Each optimization iteration then starts by a system analysis, followed by a series of subspaces optimization (possibly concurrently), where each optimization tries to solve the global problem by using approximate models of the rest of the system. After the subspaces optimizations, a full MDA is done (using only the approximate models) to perform a global optimization. The result of

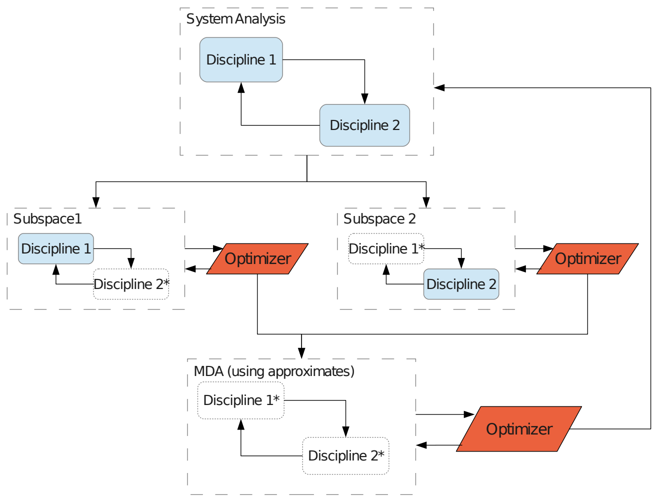


Figure 3.5: CSSO method.

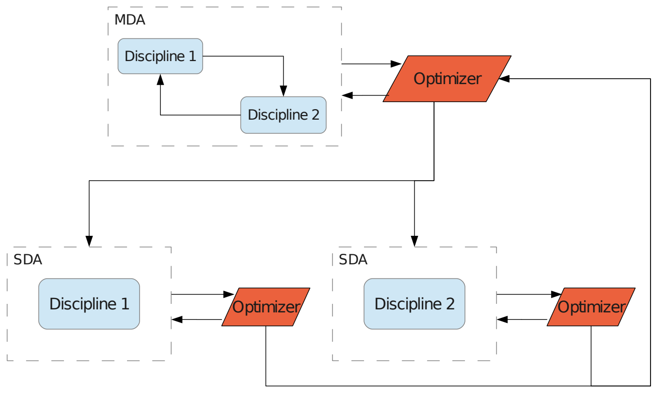


Figure 3.6: CO method.

the global optimization is then used in the next system analysis.

Originally, CSSO was developed for single-objective optimization problems. However several efforts have been made to extend CSSO to multi-objective problems (see [Ks11] for an overview of the different works in this regard).

3.2.2 Collaborative Optimization

Collaborative Optimization (CO) [Kro+94a], illustrated on Figure 3.6, reformulates the problem by replacing dependencies between disciplines by equality constraints. This transformation allows to solve the discipline-level optimizations problems in parallel. A system-level optimizer is then used to minimize the discrepancies (via the added equality constraints), while maintaining the satisfaction of the disciplines constraints.

CO is best-suited for MDO problems with a low coupling between disciplines. The authors have argued that one advantage of CO is that it closely matches the discipline decomposition of the problem, as the domain-specific variables and constraints are limited to the related disciplines. Thus, the disciplines optimization can be done by domain experts who have a strong understanding of the subproblems.

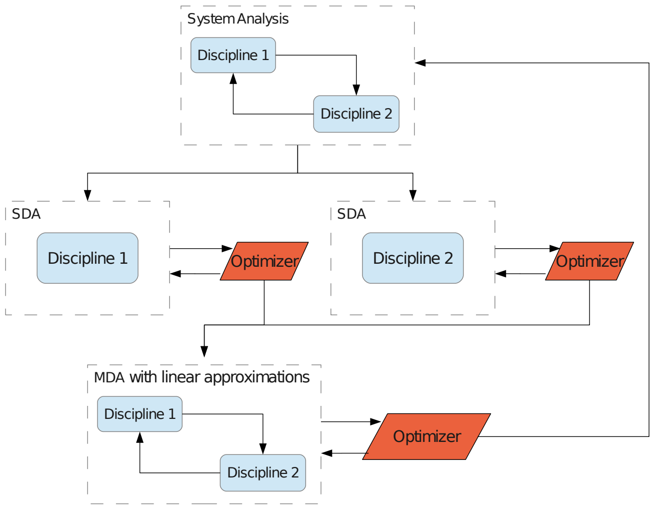


Figure 3.7: BLISS method.

3.2.3 Bilevel Integrated System Synthesis

Bi-Level Integrated System Synthesis (BLISS) [JJSRRS98], shown on Figure 3.7, has been developed to separate local and shared variables, in order to ease the distribution of the optimization process (both in regard of experts teams or computational resources). BLISS shares similarities with CSSO with the difference that local variables are assigned to the disciplines optimizations while the global variables are assigned to the global system optimization.

For each discipline optimization problem, an approximation of the global objective-functions and constraints is build, using linear approximation considering only the variables of the discipline.

The optimization process cycle alternates as follow: first a system-wide analysis is done (which includes the analysis of each subsystem) and used to provide the approximate objective-functions. Then a discipline-level optimization of the objective-functions approximations, which is used for a system optimization concerning the shared variables. These results are then used by the new system analysis at the start of the next step.

3.2.4 Asymmetric Subspace Optimization

Asymmetric Subspace Optimization (ASO), shown on Figure 3.8, is a work of Chittick and Martins [CM09] to improve on MDF for MDO problems where some disciplines are significantly more costly to analyze than the others. The classical illustration given is the one of high-fidelity aerostructural optimization, where the analysis of aerodynamics is significantly more heavier than the structural analysis. An intermediate optimization phase for the structure is introduced during the MDA, in order to reduce the number of iterations needed at the global level.

This approach can lead to significant improvements over MDF in the context of disciplines with significant analysis costs. However when the analysis costs of the disciplines are comparable, this approach is less efficient than MDF, as it introduces additional optimization steps.

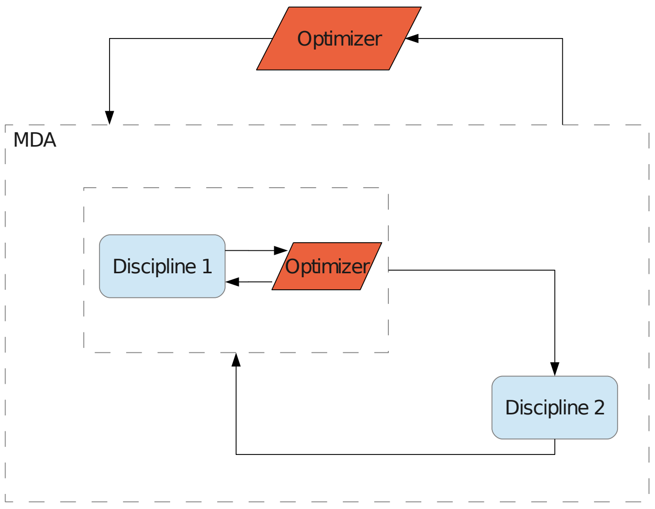


Figure 3.8: ASO method.

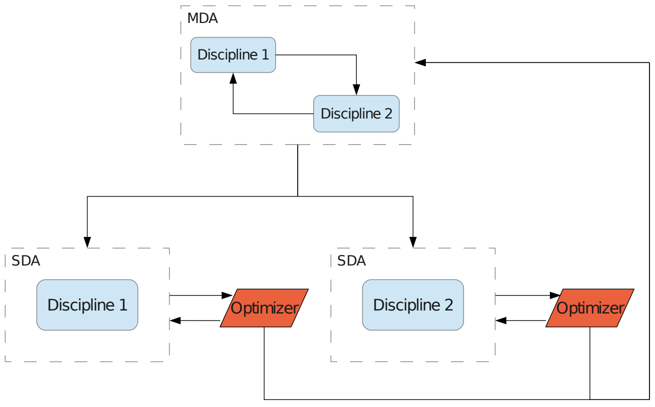


Figure 3.9: MDOIS method.

3.2.5 MDO based on Independent Subspaces

MDO based on Independent Subspaces (MDOIS) [SP05], whose representation is shown on Figure 3.9 has been developed for handling problems where the different disciplines are coupled (i.e. some outputs of one discipline are used as inputs by the others and *vice versa*) but they do not share any design variable or criterion.

MDOIS decomposes the system in separate subsystems. For each subsystem an optimization problem is defined, with its own design variables, objective-function and constraints. The coupling variables are considered constant for these subproblems. After each subsystem has solved its optimization problem, the new values of its variables are used in a system-wide analysis to be re-injected for the next iteration of subsystem optimization.

3.2.6 Quasiseparable Subsystems Decomposition

Quasiseparable Subsystems Decomposition (QSD) [HW05], represented in Figure 3.10, is another specialized method for systems which can be decomposed into subproblems which only depend on local variables and global design variables, but not on values produced by others subsystems.

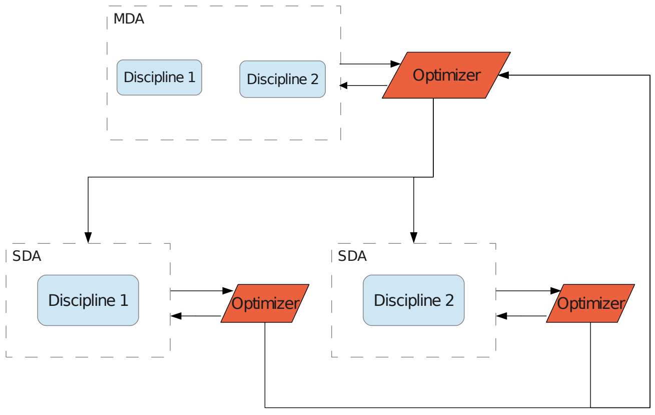


Figure 3.10: QSD method.

The basic idea is to assign to each subsystem a value for the global variables. An optimization is done on the local variables of each subsystem to maximize its constraints margins. Based on the result of each subsystem, a global optimization is done to assign new values to the global variables. The process is then repeated iteratively.

3.3 Design Optimization Under Uncertainties

When the optimization problem is used in the context of real-world applications, it is often necessary to take into account the many sources of uncertainties. While in a pure mathematical world, all models are perfectly correct and have an infinite precision, in the physical world our knowledge can be extremely limited. Moreover, high precision models can require a long time to be computed, making them prohibitively costly when used in the context of an optimization process. Finally, when the optimization problem is complex and sensitive to parameters variations, a small approximation can result in large variations of the outputs.

These difficulties are especially present in MDO problems. Not only the complexity of the disciplines themselves bring its share of modeling uncertainties related to the limited knowledge about the domains, but the product to be designed will pass through several manufacturing phases, each of them bringing its own part of uncertainties related to the production process.

In order to tackle these issues, several works have been done to take into account uncertainties into the optimization process. A major concern regarding the modeling of uncertainties is the uncertainties *propagation*. We propose to make a quick tour of the different ways which have been proposed to model and propagate uncertainties.

3.3.1 Several types of uncertainties

A distinction is usually made between *aleatory* uncertainties and *epistemic* uncertainties.

Aleatory uncertainties are inherent to the studied system. They can represent for example variability in the material used, a physical variation regarding the manufacturing of some piece, the meteorological conditions to which a device will be exposed *etc.* These uncertainties are *irreducible* as it is impossible to remove them with a better analysis of the system.

Epistemic uncertainties result from an incomplete knowledge regarding the system. These uncertainties can result from a limited set of data or lack of knowledge regarding a physical phenomenon. The uncertainties are *reducible* as it is possible to remove them with a better analysis of the system. However, removing epistemic uncertainty can often be too costly or too difficult in practice, thus they still need to be taken into account during the optimization process.

3.3.2 Uncertainties Modeling Techniques

Suppose we work on a model taking two variables as input and producing an output: $z = f(x, y)$. Based on known uncertainties on the inputs and the model, how easy is it to combine and propagate these information to determine the uncertainty regarding the output? Or more formally, can we provide a propagator P such as $u_z = P(u_x, u_y, u_f)$ (where u_i is the uncertainty associated with the element i)? As we will see, the ease to obtain such a propagator P depends on the chosen way to model the uncertainties.

We will now see several formal representations of uncertainties that have been proposed.

3.3.2.1 Probability Theory

Using the probability theory, uncertainty can be modeled using a distribution function. This modeling provides the advantages of a well-studied theoretical foundation, providing well-known combination and propagation techniques.

Aleatory uncertainties can be characterized by obtaining a distribution function from a data sample. Well-known statistical techniques can be used to see for example if a data sample follows a known probability distribution, measuring *goodness of fit* methods, that is, how well a data sample follows a given model, such as the Kolmogorov-Smirnov test [Mas51].
However, care must be taken as these techniques can introduce some more epistemic uncertainties which can lead to misleading results (for example in the case of insufficient data samples).

Concerning epistemic uncertainty, it can be more difficult to estimate a relevant distribution function, as these uncertainties represent a limitation of knowledge about the concerned factors. While aleatory uncertainties can be studied to find some good approximation function (for example by making several production experiments in order to evaluate the variations in the product), it is most difficult to do so for epistemic uncertainty.

3.3.2.2 Interval Analysis

Interval analysis can be an alternative to probability theory when the lack of information impedes modeling with a probability distribution but where the uncertainty can still be bounded within a certain domain. How easy is it to propagate intervals depends on the involved models.

For example, in the case of a monotonic function the lower and upper bounds can be determined easily. In the general case, determining the boundaries is equivalent to solve an optimization problem and can thus be done by using optimization algorithms. In the most extreme cases, one can apply sampling techniques, but this can become quite expensive. For some examples of existing techniques, one can refer to [KX08].

A limit of interval analysis is the lack of a measure equivalent to probability, which limits the usefulness of this representation in the general case. This modeling can still prove useful in the context of worst case studies where input variables can be bounded with accuracy.

3.3.2.3 Fuzzy Sets

Fuzzy sets [Zad65] can be seen as a compromise for when we still lack enough information to use probability theory, but we have more knowledge than just the bounds of the uncertainty.

Basically, fuzzy sets are sets where the membership of an element to the set is not absolute but gradual. In classical set theory, an element is or is not a member of a set. This notion can be formalized as a function $f_S : X \to \{0,1\}$ where $\begin{cases} f_S(x \in X) = 0 \Leftrightarrow x \notin S \\ f_S(x \in X) = 1 \Leftrightarrow x \in S \end{cases}$

In the context of fuzzy sets, the equivalent function can be formalized as $f_S : X \to [0,1]$, where $f_S(x \in X)$ represents the degree of membership of x to S. A value of 1 indicates a complete membership to S, a value of 0 a complete absence of membership to S and the values in between specific degrees of membership. In this regard, fuzzy sets can be seen as a generalization of classical sets.

Fuzzy sets quantification capability to represent vague information makes it attractive to model epistemic uncertainty, as it is more precise than interval analysis and well-suited to express expert knowledge. However this modeling is less powerful and expressive than probability theory, lacking for example a mean to represent an uncertainty measure equivalent to the probability of the probability theory. Indeed, the membership function is insufficient to characterize the likelihood of non-connected events.

To overcome this limitation, the fuzzy set theory was extended into the possibility theory.

3.3.2.4 Possibility Theory

Possibility theory [Zad78] seems similar to probability theory. However, they are based on axioms which diverge on a fundamental point.
The probability theory is based on the axiom of *additivity*, which says that for two disjoint sets U and V, $P(U \cup V) = P(U) + P(V)$, that is the probability of at least one of two mutually exclusive events to be verified is the sum of the probabilities of each event.

The possibility theory contains instead an axiom of *sub-additivity*, saying that for two disjoints sets U and V, $\Pi(U \cup V) = \max\left(\Pi(U), \Pi(V)\right)$ (where $\Pi(X)$ reads as "possibility of X").

Let us take the basic example of a door which can be either open or closed. If we assume "the door is closed" has a probability of 0.9, it must follow that "the door is open" has a probability of 0.1, since the sum of these two complementary events must be 1. The probability of the door to be either open or closed is $0.9 + 0.1 = 1$.

In the context of possibility theory, if we state that the possibility of "the door is closed" is 0.9, it is not incompatible with the possibility of the door to be open to be, for example, 0.4. The possibility of the door to be either open or closed is $max(0.9, 0.4) = 0.9$.

The intuition behind this difference is that probability theory applies to the reality, while possibility theory applies to the knowledge one has regarding the reality, taking into account the "fuzziness" of one's knowledge. To cite the definition proposed by Nikolaidis et al. [Nik+04]:

> *"Possibility measures the degree to which: a) A person considers that an event can occur, or b) The degree to which the available evidence does not contradict the hypothesis that the event can occur."*

As well as the notion of *possibility*, possibility theory introduces the notion of *necessity*. Basically, $Nec(U) = 1 - \Pi(\bar{U})$. This definition brings several interesting properties:

$$\begin{cases} Nec(U) \leq \Pi(U) \\ Nec(U) + \Pi(\bar{U}) = 1 \\ \Pi(U) + \Pi(\bar{U}) \geq 1 \end{cases}$$

Necessity and possibility of an event e can be viewed as lower and upper bounds to the probability of e.

Possibility theory offers numerous tools similar to the ones of probability theory (so much in fact that it has been debated if the two theories are really different, or if possibility theory was merely a variation on probability theory). However, the capability of this theory to model expert knowledge has made it quite popular in the context of uncertainty modeling.

3.3.2.5 Evidence Theory

Evidence theory [Sha76], also known as Dempster-Shafer theory (DST), takes into account available evidences to provide a degree of belief concerning a fact.
The basic idea of this theory is to represent the notion that the more evidences seem to confirm a proposition, the more we can believe the proposition to be true.

Evidence theory represents a proposition S as a set of elements. Thus some propositions can include others propositions (following the basic set inclusion definition). To these sets are assigned a basic belief assignment (BBA), also called mass.

From this mass can be calculated two measures:

- ▷ *Belief:* $bel(S) = \sum_{S' \subseteq S} mass(S')$
- ▷ *Plausibility:* $pl(S) = \sum_{S' | S' \cap S \neq \emptyset} mass(S')$

The mass measurement represents the amount of evidences which support the proposition, the likelihood of S. The belief and plausibility can be seen as lower and upper bounds to this likelihood and, as with possibility theory, can also be used as lower and upper bounds for probability.

Once again we can obtain some interesting properties regarding these measures:

$$\begin{cases} bel(S) \leq mass(S) \leq pl(S) \\ pl(S) = 1 - bel(\bar{S}) \\ bl(S) + bl(\bar{s}) \leq 1 \\ pl(S) + pl(\bar{s}) \geq 1 \end{cases}$$

Several rules have been proposed to combine informations coming from different (potentially conflicting) sources. To see an overview of these proposals, the reader can refer to [SF02].

3.3.2.6 Analysis of Uncertainties Modeling

We have seen how different ways to represent the uncertainties have been proposed. Some of these methods are based on well-studied mathematical tools, such as probabilities theories, but may not be adapted to the lack of quantifiable information. To palliate this insufficiency, alternative representations have been developed, based on the expression of expert knowledge or of available evidences.

In a similar way to optimization methods, the choice of the uncertainties representation to apply will depend on the context, on the sources of uncertainties to take in account and the proficiencies of the experts.

3.3.3 Using Uncertainty for Robust Optimization

On Figure 3.11 a function with two optima can be seen. The minimum labeled a is only a local optimum, being less optimal than the global minimum b. However, a is more robust than b, in the sense that its neighborhood does not change drastically. A variation in the inputs, or in the actual results compared to the analytical model, will result in a lesser degradation of performances. Consequently, it can be deemed preferable to select a instead of b, if the uncertainties would possibly result in a worst solution when trying to achieve the solution b.
Specific methods have been dedicated to find solution which are both good (local optimum) and robust.

3.3.3.1 Taguchi Method - The first steps of Robust Optimization

Robust optimization tries to provide a solution which is both good and insensible to small variations of the inputs.

The research on robust optimization has been initiated with Taguchi's robust design methodology[Tsu92], aiming at improving the quality of manufactured goods.

In his methodology, Taguchi proposes a three-stages process:

- ▷ System design, where the designers determine the overall structure of the product at a high conceptual level.
- ▷ Parameter design, where the optimal values of the design variables are determined.

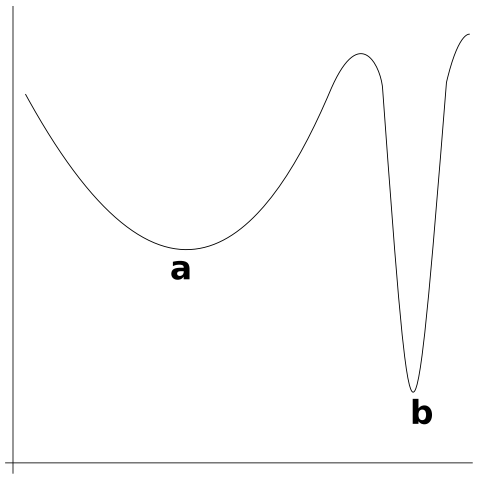


Figure 3.11: Example of robust optimum (a) and non-robust optimum (b).

▷ Tolerance design, which focuses on reducing the variability of the various parameters to fix an acceptable limit to the variability of quality for the product.

To help the designers during the parameter design phase, Taguchi introduces several measures, among them the Signal-to-Noise (SN) ratios. These ratios are used to estimate the sensitivity of a performance of the product to variations. Each ratio relates to a possible goal regarding the studied performance: larger the better, smaller the better, on target the best. These ratios are respectively noted SN_L, SN_S and SN_T.

By simulation or experimentation, one must first produce a data set.The SN_L and SN_S can be directly calculated as

$$SN_L = -10\log\left(\frac{1}{n}\sum_{i=1}^{n}\frac{1}{y_i^2}\right)$$

$$SN_S = -10\log\left(\frac{1}{n}\sum_{i=1}^{n}y_i^2\right)$$

For SN_T, we need first to measure the mean response, given by $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ This mean can then be used to calculate the standard deviation as follows

$$S = \sqrt{\sum_{i=1}^{n}\frac{(y_i - \bar{y})^2}{n-1}}$$

SN_T can then be calculated as

$$SN_T = 10\log\left(\frac{\bar{y}^2}{S^2}\right)$$

To reduce the sensitivity of the solution to noise, the SN ratio must be maximized. To this end, Taguchi uses Design of Experiment[Sac+89], a statistical procedure for determining the effects of multiple inputs on a desired output. By evaluation different designs using different noise factors (temperature, pressure *etc.*), it is possible to obtain an array of SN rations on which a statistical data analysis allows to identify which design provides the best performances.

Tagushi was the first to propose a way to take in account the uncertainties in design. As it is nearly inevitable for such pioneering work, his approach suffers from several limitations. As the number of parameters increases, so do the number of experiments (for N parameters and M noise factors we need 2^N designs and $M2^N$ experiments). Moreover, the statistical measures proposed by Tagushi has been subject to much debate (for a discussion of these measures see [Nai+92]). However, Tagushi opened the way for the other methods in the field.

3.3.4 Uncertainties in Multidisciplinary Optimization

The modeling of uncertainties in MDO problem is not fundamentally different from any other optimization problems. As with the other aspects of optimization, it is the complexity of the problem which makes classical uncertainty propagation methods unfeasible in practice as well as the heterogeneity of formalisms brought by different teams working in the different domains of the problem.

Doing a global uncertainty analysis on a MDO problem can be even most costly than a system analysis. Several strategies have been proposed for the propagation of uncertainties. Many of them suppose an uniform representation of uncertainties (for example [DC05; Liu+06] assume the uncertainties are modeled by probabilistic normal laws, while [Gu+00; LA08] use interval definition). Whatever the strategy, propagating uncertainties will require costly additional evaluations of the disciplines, and often of the entire problem. This statistical approach for taking into account uncertainties makes an already costly problem even more expensive to evaluate [Koc+99]. It is often necessary to fall back on approximate models in order to reduce the computational cost of evaluating uncertainties [All+06]. Of course the drawback is that the uncertainties evaluation will be more imprecise on these approximate models.

We have already noted that an additional and somewhat unique challenge of MDO comes from the need to integrate heterogeneous disciplines potentially coming from different experts teams with different concerns. Surprisingly, while this fact have been put in front to justify the usefulness of multi-level MDO methods, its implications regarding the uncertainties manipulation are not so much discussed. We already saw in the previous chapter that multiple concurrent uncertainties representations have been developed, each one with its advantages and drawbacks. Without doubts different experts would have different preferences in regard of which modeling to chose. However current propagation techniques require a homogeneous uncertainties representation for the problem.

3.4 Analysis of MDO

MDO methods try to address the problem of complex continuous optimization problems. The principal difficulty of such problems is not their large size, but the inherent coupling and interrelationships between the different parts, as well as the costly and heavy computational models they involve. These properties make classical optimization techniques inefficient.

MDO methods propose to reduce the complexity of the problem by separating it into different *disciplines*. While the dividing strategy can vary depending on the method, the main idea is to identify loosely coupled "blocks" in the problems, which are independent enough to be separated without impacting too much the solving process, and which are simple enough to be solved using known optimization techniques.

This transformation of a highly coupled system to a loosely coupled one cannot fail to remember the comparison of Wilden [Wil03] between "hot" (complex) and "cold" (complicated) systems, the first ones being highly connected and integrated networks, the second ones loosely integrated and mostly in a tree-like structures (we will indeed see in the next parts how this representation of the system as an integrated network of components is relevant).
By reducing the complexity of the problem, MDO methods indeed ease the optimization process, but the price of this transformation is the necessity to include intermediate steps in order to re-establish a coherent "view" of the whole system. By taking a reductionist approach, MDO methods are bound to suffer for this additional cost, requiring multiple back-and-forth due to their maimed representation.

On a side note, we can wonder about the existence of inherently identifiable, separable disciplines. One can argue that this distinction between disciplines does not correspond to a natural separation inherent to the system to design, but is a consequence of the current division on experts into separate expertise fields. Is the fact that an airplane design problem can be decomposed between aerodynamic, geometry, *etc.* an inherent property due to the nature of the problem? Or is the reason that this problem was constructed in the first place by aggregating the work of experts coming from these fields? In this context, it is not really surprising that the disciplines corresponding to these fields will be strongly internally connected and only weakly connected to the others. And on another hand, it is easy to imagine how experts from these different fields will tend to favor a divide into disciplines which correspond to their specific expertise field.
Consequently, by maintaining this division between potentially arbitrary disciplines, MDO methods are bound by the same reductionism which led to the distinction between these disciplines in the first place.

Beyond this fundamental limitation, one major shortcomings of these methods is the heavy work and expertise they require from the engineer to be put in practice. To actually perform the optimization process, one must have a deep understanding of the models involved as well as of the chosen method itself. This is mandatory to be able to correctly reformulate the models according to the formalism the method requires, as well as to work out what is the most efficient way to organize the models in regard to the method. Since by definition MDO involves disciplines of different natures, it is often impossible for one person to possess all the required knowledge, needing the involvement of a whole team in the process. Moreover, answering all these requirements implies a lot of work *before* even starting the optimization process.

Interestingly, different comparisons of multiple methods [PLB04; YSP08] showed that complex, multi-level methods (BLISS, CSSO...) did not consistently outperform more "simple" mono-level methods (IDF, AAO...) in regard of optimization performances (quality of solution, number of evaluations). In each case, the authors advocated that the advantages of complex MDO methods were more about an "organizational efficiency", as these methods are more flexible and can more easily be adapted to existing organizational structures. While the ways to measure such properties can make such analysis debatable, this argument illustrates an important point about MDO methods: the complexity of the problem to solve is such that the main goal is not so much to efficiently solve it than to provide experts with simple and adapted tools in order to explore it. Sadly, the same evaluation works concluded that the more portable and flexible solutions were also the more difficult to put in practice.

Consequently, it seems that MDO methods suffer from a gap, not in regard to their computational efficiency (which seems nearly to be a secondary goal) but from the lack of a method which would be both flexible and simple to put in practice.

Conclusion on Optimization

We have seen how different types of optimization problems have been defined over time, depending on their topology, their complexity, their specificities.

Something worth noticing is the fact that all these types of problems are not *inherently* different, but share a common structure. Indeed, a mono-objective optimization problem is just a special case of a multi-objective problem. And a multi-disciplinary problem is basically an optimization problem so complex that standard optimization techniques fail. These differences are in no way indicative of a fundamental distinction between these different kinds of problems, but come solely from the limitations of the existing optimization techniques, which have to choose between being applicable in the general case and being efficient.

An interesting observation concerning MDO techniques is the fact that, to evaluate their performances, they are sometimes applied to simple optimization problems, solvable by classical optimization techniques (see for example [Kro+94b] for an application of CO to the Rosenbrock's valley problem). This observation illustrates the fact that all optimization problems share a common structure and differ only in their complexity (as already stated by the quote from Martin and Lamb mentioned in the beginning of chapter 3). Of course these examples are usually only used as illustration, as MDO methods are too heavy to be interesting to use for such problems.

These conclusions raise an interesting question: could it be possible to create an optimization technique which would scale from simple optimization problems to complex ones?

In itself this statement seems to be contradicted by the intuition provided by the NFL theorems. However we have also seen how at the extreme end of complexity, the concern is not so much about finding an efficient optimization technique, but finding an efficient *organization* to apply specialized optimization techniques to the different parts of the problem, while keeping a global coherence.
At this level, the actual optimization processes can be abstracted as black boxes, which are handled behind the scene by experts or automated processes.

Thus we can reformulate our question as: could it be possible to create a technique which would provide an adequate organization for each type of problem, adapted both to simple problems which need to be solved quickly and to large-scale optimization problem involving whole disciplines?

Currently, only MDO techniques could be seen as being applicable to the whole range of

optimization problems, but their strict structure make them too cumbersome for such a task and seems to suffer from a trade-off between simplicity and flexibility.
An ideal solution for such a problem would be a method able to adapt itself to the problem at hand, in order to scale with the needs of the engineers.

The next part of this thesis concentrates on providing such a method.

4 Multi-Agent Systems for Optimization and the AMAS Theory

As stated in our conclusion on optimization, providing a method able to scale to the needs of the full range of optimization problems will requires it to be capable of adapting to the problem at hand.

The main theme of the SMAC team[1], in which this thesis has been realized, is the Adaptive Multi-Agent Systems (AMAS) Theory. This theory relates to the design of agent-based complex systems with self-adaptive capabilities.

In this chapter we will first describe what multi-agent systems are and how they can be used for problem solving, before concentrating on the concepts of the AMAS theory.

4.1 Multi-Agent Systems

Multi-Agent Systems (MAS) is a relatively recent field which can be seen as the intersection of Artificial Intelligence (AI) and Systems Theory.

As a reminder, the AI field was developed in 1950s as "the science and engineering of making intelligent machines", as stated by McCarthy, one of the pioneers of the field [McC+06]. This rather ambitious project was somewhat toned down during the 1970s when the field was the subject of several setbacks leading to an "AI winter" [Hen08], whose effects can still be felt today. The commonly accepted reason for this setback was that researchers had been too optimistic in their expectations of the breakthroughs which would be produced by the field, and did not take enough into account the inherent complexity of some of the tasks they were proposing to handle (*e.g.* language processing).

This disgrace period of the AI field ended with the success of expert systems in the 1980s. These systems aim to emulate the ability of a human being to take decisions based on expert knowledge, using inference mechanisms (via an *inference engine*) and a rules database. However, even expert systems cannot avoid the complexity of modeling knowledge, and are still ultimately limited by the growth of their rules database. This concern, among others (such as privacy of informations) led to a new field of AI named Distributed Artificial

[1]*Systèmes Multi-Agents Coopératifs* (Cooperative Multi-Agent Systems)
`http://irit.fr/-Equipe-SMAC-`

Intelligence (DAI) [OJ96], where several expert systems collaborate to provide a collective diagnostic of a situation.

In parallel to the developments of AI, another field of knowledge emerged in the beginning of the century, Cybernetics (also called System Theory), the study of self-regulating systems. Interestingly, this field had radically different origins from AI, taking root in social and natural sciences. These two disciplines had a somewhat uneasy coexistence for some times during the 50s, after which AI took the lead and cybernetics was somewhat relegated in the background (on this topic, see for example [Car10]). The field achieved a revival in the 1970s with the "new cybernetics", or "second-order cybernetics", which introduces the study of self-organizing systems and the notion of external observer.

It is interesting to note the conflicting nature of AI and cybernetics. AI initially based itself on a reductionist approach of knowledge, using symbol manipulation coming from algebra and logics. Cybernetics was part of the more general epistemological upheaval of Constructivism.

It is at the conjunction of these two seemingly contradictory fields that was born the study of Multi-Agent Systems.

4.1.1 Principles of Multi-Agent System

Before talking about MAS, we must explain the notion of *agent*. Several definitions of what is an agent have been proposed. We keep here the (mostly) consensual one proposed by Wooldridge in [Wei99]:

Agent - "An *agent* is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives."

based on this definition, a MAS is a system composed of several agents, interacting among each others and with their environment.

The *autonomy* of an agent is the fundamental characteristic that differentiates it from, for example, the computer science concept of object. While an object is a passive entity encapsulating some data and functions, waiting to be solicited, an agent is capable of acting and reacting with its environment. From this comparison it should be clear that the concept of agent is, like the concept of object, the building brick of a paradigm which can be used to model a complex reality. And indeed, agents have been used in a great variety of fields, a fact which can contribute to explaining the difficulty to produce an unified definition of the concept.

4.1.2 Self-* capabilities

While it is not true for all MAS, some interesting properties can be achieved when taking advantage of the autonomy of the agents. This autonomy, coupled with an adequate behavior of the agents, can lead to systems able to adjust, organize, react to changes *etc.* without the need for an external authority to guide them. These properties are regrouped under the term self-* capabilities (self-*tuning*, self-*organizing*, self-*healing*...).

Not all MAS necessarily present all of these self-* capabilities but, as a result of building a system from autonomous and locally situated agents, many MAS will exhibit them to some degree. Consequently, MAS are often relevant for dynamically taking into account changes in their environment. For example, a MAS in charge of regulating the traffic of packets in a computer network could be able to react efficiently to the disappearance of some of the relay nodes.

4.1.3 Multi-Agent Systems for Distributed Problem Solving

In the context of this thesis, we will concentrate on the application of MAS in the specific context of Distributed Problem Solving (DPS). However it can be useful to bear in mind the others possible application fields: social simulation, biological modeling, systems control, robotics *etc.* and agent-oriented modeling as a programming paradigm in general.

4.1.3.1 Multi-Agent Systems for Combinatorial Optimization

MAS has been applied with great success to multiple combinatorial optimization problems. Many application fields have been proposed, among which "smart grids" power systems control [Roc+13], sensors networks [VRAC11], supply chain management [Ada+11; Kad11] *etc.*

A major part of the literature on the application of MAS to combinatorial optimization concerns Distributed Constraint Satisfaction Problems (DCSP, or DisCSP) and its extension, Distributed Constrained Optimization Problems (DCOP, or DisCOP).

DCSP [Yok+98] is a formalism to model Constraint Satisfactions Problems (CSP) using agents. A CSP is defined as a triplet $<X, D, C>$ where:

- ▷ $X = x_1, ..., x_n$ is the set of variables.
- ▷ $D = D_1, ..., D_n$ where D_i is the definition domain of x_i.
- ▷ $C = c_1, ..., c_m$ the set of constraints to satisfy.

The goal is to find an assignment to the set of variables X which:

- ▷ comply with their definition domains D.
- ▷ satisfy the set of constraints C.

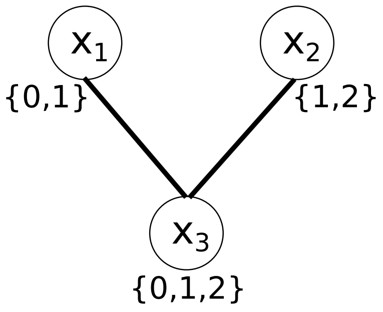
In order to simplify the representation, the constraints of a CSP are often binary. In this case, the CSP can be represented as a graph where each vertex is a variable and each constraint an edge.

From this graph representation, the DCSP formalism models a CSP as an agent graph, with each agent in charge of assigning a value to a variable based on local and shared constraints. A DCSP is described as a quadruplet $<X, D, C, A>$. X, D and C have the same meaning than in the CSP formalism, while A is the set of agents. Each agent in A has the responsibility of a subset of X and knows the constraints related to the variables in its care. The most common assumption is that each agent has the responsibility for one and only one variable. See Figure 4.1 for an illustration of the graph transformation of a DCSP.

One classical example of DCSP solver is Asynchronous Backtracking (ABT) [Yok01]. ABT creates a total ordering on the agents and models the relations between agents as directed

$$X = \{x1, x2, x3\}$$
$$D = \{\{0,1\}, \{1,2\}, \{0,1,2\}\}$$
$$C = \{(x1 \neq x3), (x2 \neq x3)\}$$

(a) formal definition.



(b) corresponding agent graph.

Figure 4.1: An example of the DCSP representation.

links, in order to make the network cycle-free. In ABT agents exchange *nogoods*, which are conditional constraints (the constraint is valid as long as the other agents do not change states). When an agent detects a *nogood*, it checks if it can change its state in order to solve it. If it is possible it does so and informs the agents to which it is linked. Else it propagates it to the lowest priority agent it knows (based on the established ordering) involved in the *nogood*, creating new links if necessary.

An extension of DCSP has been proposed for formalizing Distributed Constraint Optimization Problems (DCOP, or DisCOP). DCOP is to COP the equivalent of DCSP to CSP. While a DCOP is described in the same way than a DCSP, the semantic and goal are different. In DCOP, the constraints in C represent a decomposition of a global cost function, which the agents try to minimize (or alternatively, maximize). Each constraint is now seen as a local cost function giving the cost associated with each state of the involved variables (in the context of DCOP, the term *constraint* is sometimes replaced by the terms *cost function* or *soft constraint*). Formally, DCOP considers the global objective-function $F(X) = \sum_{x_i, x_j \in X} c_{ij}(x_i, x_j)$ where c_{ij} is a local cost function associated with the states of x_i and x_j.

A classical use of DCOP is unsolvable CSP, problems where there is no solution which satisfies all the constraints. In these cases the problem can be changed into a DCOP with the new objective to minimize the number of violated constraints. If we wanted to transform the problem shown in Figure 4.1 (ignoring the fact that this DCSP is in itself solvable), we could replace the constraints in C by the function c_{13} and c_{23} defined as follows:

x_1	x_3	$c_{13}(x_1, x_3)$
0	0	1
1	0	0
0	1	0
1	1	1
0	2	0
1	2	0

x_2	x_3	$c_{23}(x_2, x_3)$
1	0	0
2	0	0
1	1	1
2	1	0
1	2	0
2	2	1

In the context of this framework, many techniques have been proposed. One of the leading algorithms to date is the Asynchronous Distributed Optimization Algorithm (ADOPT) [Mod+06].In order to work, ADOPT also needs to reformulate the problem by introducing a total ordering on the agents. This ordering allows to create a tree representation of the problem, where each agent has a single parent and multiple children. The algorithm in itself is articulated on two main ideas:

- ▷ each agent asynchronously changes states when it perceives a possible better solution.
- ▷ the agents can backtrack to previously explored solutions, but only if this current state is worse than a specific *backtrack threshold* determined by its parent agent.

Another well-known algorithm is Optimal Asynchronous Partial Overlay (OptAPO) [ML04]. This algorithm is an adaptation of Asynchronous Partial Overlay (APO), which was proposed for solving DCSP. OptAPO, as APO, is based on the principle of *Cooperative Mediation*, where the agents try to identify parts of the problems which can be solved in a centralized way (using centralized solvers such as Branch-and-Bound). During solving, some of the agents take the role of mediators during *mediator sessions*, where they compute a solution to a part of the whole problem and propose the solution to the others agents involved into the mediation.

The DCOP formalism is very popular in its own right as it allows a clear framework on how to represent this type of combinatorial problems without putting too much restrictions on how the problem is solved. The exact information shared by the agents, and the way they communicate among themselves is not constrained. However this formalism is not without limitations.
While some works successfully used DCOP in the context of continuous optimization[Str+09], this formalism is not adequate to handle the full range of continuous optimization problems. DCOP was conceived for a specific type of problems where the difficulty resides in the combination of multiple constraints. These problems are supposed to be easily decomposable into several cost functions, where the cost values associated to the variables states are supposed to be known. This major assumption does not stand for complex continuous optimization problems (such as MDO problems for example), where the complexity of the models and their interdependencies cause this information to be unavailable in most cases.

It is interesting to note that many methods require non-trivial changes to the topology of the agent graph to work (acyclic graph, tree structure...), both in the context of DCSP and DCOP. These changes can be potentially extensive operations in themselves, and must be done carefully lest the relevance of the results be compromised. In this regard, most existing agent-based optimization techniques for DCOP may require a strong expertise to be efficiently applied[Kad11].

4.1.3.2 Multi-Agent Systems for Continuous Optimization

While MAS are a popular approach for solving combinatorial optimization problems, their application to continuous optimization is scarce at best. One could explain this discrepancy by the fact that continuous optimization problems are in general more difficult to decompose than combinatorial ones.

As said in the previous section, an adaptation of DCOP with continuous variables has been proposed in [Str+09]. This work proposes an adaptation of the max-sum algorithm in the continuous case by redefining the two operations of summation and maximization in order to be able to use them over continuous utility functions. However, this work concentrates on a specific type of problem (distributed sensor networks) and the utility functions must be piecewise linear. It is not applicable to the broader scope of general continuous optimization problems.

Some works have been proposed for using MAS for dynamic continuous optimization (see [Lep+10] for example). However these works usually involve population-based exploration of the search space, where one agent represents a single candidate solution of the problem. These kinds of population-based approaches are ill-suited for complex problems, where a major difficulty is the cost of evaluating a point in itself.

A notable work on the subject of complex continuous optimization is the MASCODE algorithm [Wel+06], which concentrates on MOO and MDO problems. In MASCODE, each agent is in charge of a discipline and the links between agents represent the dependencies between the different disciplines. For each input of a discipline, a *physical validity interval* and an *objective validity interval* are defined. These two intervals represent respectively the physical constraints of the model and the boundaries of an objective to achieve. Using these two measures, a satisfaction indicator is defined using a parametric piecewise continuous function. The basic idea of this satisfaction indicator is as follows:

▷ when the value of the input is in the boundaries of the objective validity interval, the satisfaction of the agent regarding this input is high,
▷ when the value of the input is outside of the objective validity interval but inside the physical validity interval, the satisfaction lowers,
▷ if the value of the input goes outside the physical validity interval, the satisfaction becomes minimal.

The agents use the values of their inputs to send *forward messages*, informing others agents of the values of their outputs. Upon reception of these messages, an agent uses these new values to recalculate its own outputs (potentially sending in turn new *forward messages*) and send *backward messages* to the agents controlling their inputs. These *backward messages* are modification requests indicating the satisfaction of the requesting agent. Upon reception of such messages, the agent will select which ones to handle based on the current satisfaction degree of their sender, and will change the value of its inputs accordingly (sending in turn new *backward messages* if required).

This algorithm is very interesting as it makes very few presuppositions concerning the shape of the optimization problem. The problem can be expressed as initially conceived by the designer, without requiring special transformation operations. Still, there are some limitations to the expressiveness of the formulation. The possibilities for the designer to express constraints and objectives of the problem are restricted to the specific use of the physical and objective validity intervals. As a consequence, constraints and objectives cannot be expressed independently, and can only concern *one* variable at a time. While this latter limitation can somewhat be circumvented by introducing artificial disciplines placeholders, the natural modeling of the domain is then lost. This limitation can be explained as a consequence to the specific application domain which was considered for the algorithm (industrial product design).

4.1.3.3 Analysis of Multi-Agent Systems for Distributed Problem Solving

We have seen in this section how MAS have been applied for problem solving. While the solving of discrete constraint satisfaction and optimization problems has been a very popular topic of interest, very few works exist concerning the solving of continuous problems. Among the few existing works, most concentrate on specific application topics, thus no real unified effort exist in this regard comparable to one which can be observed for discrete optimization.

One of the main contributions of this thesis is to rectify to this deficiency by providing not only an agent based method for general continuous optimization but also a general modeling for enabling further contributions on a comparable basis.

4.2 The Adaptive Multi-Agent Systems Theory

4.2.1 Theorem of Functional Adequacy

The design of a MAS for problem solving is not an easy endeavor. We can observe that many of the MAS for problem solving proposed by the scientific community are nature-inspired (ants, flocks, bees, bats etc.) [DMSGK11]. Indeed, Nature had a long time to experiment on several arduous problem and plenty of subjects at hand. Why would we not take advantage of that? However, such a strategy presents a severe limitation, as it is ultimately restricted into the potential solutions it can provide.

In order to overcome this limitation, we need to rely on a more theoretical approach which would help us in the design of MAS for which we do not know any existing applicable mechanisms. Such an approach is provided by the Adaptive Multi-Agent Systems (AMAS) theory.

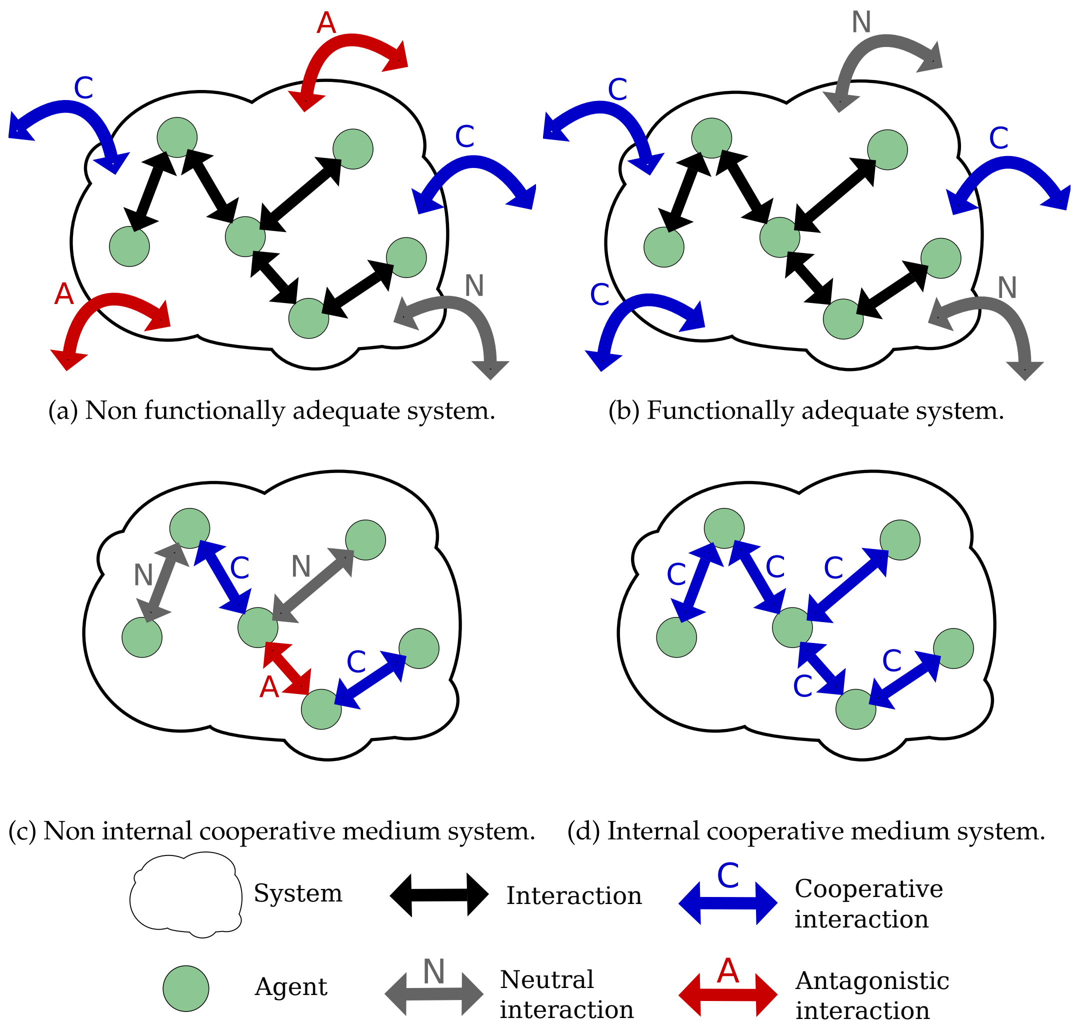
The AMAS theory was developed by the SMAC team and formalized in [Gli01]. It focuses on cooperation as the fundamental mechanism of MAS design.

At the base of the theory is the modeling of a system as a set of entities, the agents, interacting with each others and with their environment. The system can be deemed to be *functionally adequate* by an external observer if this latter judges that the system as a whole correctly accomplishes its function in regard to the environment. This external observer can be considered to be a perfect oracle, in a way similar to the Laplace's demon, knowing exactly what will be the consequences of the interactions between the system and its environment.

It is important to understand that, from a theoretical point of view, the notion of functional adequacy is inherently subjective, and depends on the observer. In practice it is however easier to attain a reasonable consensus. For example a natural system will usually be deemed adequate if it survives and thrives in a sustainable way. For an artificial system it is often even easier since the functional adequacy corresponds to the function expected by the designer of the system.

The AMAS theory identifies three categories of interactions between a system and its environment:

- ▷ Cooperative action: the acting entity is beneficial to the other.
- ▷ Antinomic action: the acting entity is detrimental to the other.



(a) Non functionally adequate system. (b) Functionally adequate system.

(c) Non internal cooperative medium system. (d) Internal cooperative medium system.

Figure 4.2: Illustration of functionally adequate and internal cooperative medium systems.

▻ Neutral action: the acting entity has no effect on the other.

From this categorization, the theory draws its formal definition (and fundamental axiom) of functional adequacy:

Axiom of functional adequacy - A functionally adequate system has no antinomic interaction with its environment

An illustration this axiom is shown on Figure 4.2a and Figure 4.2b.

Using this axiom, several properties have been demonstrated concerning the specific set of *internal cooperative medium systems*, defined as systems in which the agents do not have any antinomic or neutral interaction (illustrated on Figure 4.2c and Figure 4.2d). We will not enter here in the details of these properties and their demonstration, the interested reader can refer to [Gli01; GCG99]. Suffice to say that these properties lead to the central theorem of functional adequacy:

Theorem of Functional Adequacy - For each functionally adequate system there exists an internal cooperative medium system also functionally adequate in the same environment

Violated condition	**Corresponding NCS**
$C_{perception}$	Incomprehension, Ambiguity
$C_{decision}$	Incompetence, Unproductiveness
C_{action}	Uselessness, Competition, Conflict

Figure 4.3: The conditions for cooperation and corresponding NCS.

This theorem is at the core of the AMAS approach of system design. We already know that for each problem there is possibly an infinity of equivalent systems producing the same adequate functioning. Using the theorem of functional adequacy we can concentrate on designing an internal cooperative medium system, that is, designing a system were the agents cooperate among themselves and with their environment. The goal of a designer using this approach is thus to study the nature of the interactions between the entities of the problem domain, and to see how the non cooperative situations could be corrected in order to obtain an internal cooperative medium system.

In addition to providing us with a theoretical context, this approach gives us another interesting property: as the design of the system is focused on the local interactions of the agents, we do not need to explicitly take into account the global function of the system. This property is extremely significant. If the global function is complex, it can be extremely difficult to successfully design a system which explicitly tries to achieve the global function (top-down approach). By concentrating on the local functions of the agents, we can spread the complexity and ease the design of the system (bottom-up approach). As such, this approach strongly relies on emergence phenomenons and is often referred to as "Emergent Problem Solving" [Qui00].

4.2.2 Cooperative Agents and Non Cooperative Situations

To guide the designer during the building of such internal cooperative medium system, the conditions of what makes an agent a *cooperative agent* have been further formalized. An agent is said to be cooperative if it satisfies three conditions:

- $C_{perception}$ Every perceived signal can be understood without ambiguity.
- $C_{decision}$ Every interpretation must produces useful information.
- C_{action} Every action done based on the decision must be useful.

Based on these conditions, a set of *Non Cooperative Situations*(NCSs) has been identified. These NCSs correspond to interactions which are not cooperative, and must be removed for the system to be an internal cooperative medium system. The NCSs are classified based on the condition they violate. The table in Figure 4.3 shows this classification.

The different NCSs are:

- **Incomprehension** The agent is not able to extract information from a received message.
- **Ambiguity** The exact meaning of a message cannot be determined, or lacks required informations.
- **Incompetence** The agent does not have the capabilities to handle a received information.
- **Unproductiveness** A received information does not lead to any useful conclusion.

- ▷ **Conflict** The action of the agent is incompatible with an action from its environment.
- ▷ **Competition** The action of the agent leads to the same result than an action from its environment.
- ▷ **Uselessness** The action of the agent has no effect on itself or its environment.

A cooperative agent actively tries to avoid these NCSs and, should this fail, to solve them to the best of its capabilities. To this end, three distinct mechanisms can be used[BBG09]:

- ▷ **1. Tuning** The agent can change one or several of its internal parameters (*e.g.*, adjusting the priorities of its behavior rules).
- ▷ **2. Reorganization** The agent can change its relationship with its environment (*e.g.*, removing or creating new links with others agents).
- ▷ **3. Evolution** The agent can change the nature of its environment (*e.g.*, removing or creating new agents).

The order of these mechanisms usually correspond to their level of disruption (*i.e.*, adjusting its parameters usually has less consequences than creating and removing agents). In general, it is preferable to make the less disruptive possible adjustment. One can for example design the agents based on an escalation principle, where the agents try to solve a NCS first by using tuning, escalating to a more disruptive mechanism only when the previous ones failed to solve the NCS. Of course, a NCS situation being by itself disruptive, it is sometime more efficient to immediately make a more radical adjustment in order to solve the NCS more quickly. The designer will have to balance these concerns according to the specificities of the system.

4.2.3 The Importance of Locality

A primordial aspect of the AMAS theory is the importance of locality. The theory insists on the need to consider and consider only the partial knowledge and local interactions between the agents, without trying to provide a "bigger picture". While this concern can be linked to specific notions like the concept of emergence, we believe that a direct explanation can be found regarding preoccupations about the *scalability* of the system.

The AMAS theory concerns systems which aim to solve complex problems. By definition the difficulty of such problem increases exponentially with its size. To design a MAS able to scale with the size of the problem, the designer has in general two possibilities: he can increase the size of the system or increase the complexity of the agents. The key difference between these two operations is their marginal costs.
While adding a new agent to the system should usually represent a constant cost (in terms of complexity, computational requirements ...), increasing the complexity of the agents will be more and more difficult, since the agents will individually reach the same limitations as centralized methods. The principle of locality is a good example for this argument. Suppose a hierarchical system where one of the agent is in charge of a whole subsystem. If the increase of the complexity of the problems results in an increase of complexity of the subsystem, after some limits the agent will not be able to handle correctly the subsystem, resulting in a limit to the scalability of the system as a whole.

The software engineer will not miss the uncanny similarity of this argument with the relatively recent trend regarding scalability concerns for computer infrastructures (for *e.g.*

web servers, distributed databases *etc.*). The two main categories for scaling resources in such systems are *vertical scalability* (scaling "up") and *horizontal scalability* (scaling "out"). Vertical scalability is the improvement of existing resources for them to be able to handle more data/traffic/..., while horizontal scalability consists in adding more resources to spread the workload. While in the past vertical scalability was the dominant practice, the current consensus seems to be that horizontal scalability is easier and provides better performances increase [Mic+07].
Of course such comparison must be made with caution, as the context of the two fields are quite different. However it is interesting to note how similar concerns from these different fields led to a similar evolution in the approaches.

Obviously, such general principles cannot hold systematically true and some problems which cannot be solved by adding more agents are easily resolved by improving their reasoning capabilities. More so, in some cases increasing the size of the system can increase the complexity of the existing agents (for example by adding neighbors to an agent, thus increasing the complexity of its decision process). Still, in the general case, the motto of approaches such as the one of the AMAS theory could be "scale out when you can, scale up when you must".

4.2.4 ADELFE - A Method for Designing AMAS

ADELFE [Ber+03] is a method dedicated to the development of AMAS. The name "ADELFE" is the French acronym for "toolkit to develop software with emergent functionality" (*Atelier pour le DEveloppement de Logiciels à Fonctionnalité Emergente*). While ADELFE is not the only method devoted to guide the design of a MAS, it is the only one specifically tailored for AMAS.

The ambition of ADELFE is to provide be-all and end-all method to guide engineers during all the phases of the design of an AMAS, from the high-level requirements to the "nuts and bolts" implantation details. This ambition was the driving factor for multiple projects with the objective to improve or complement ADELFE with additional tools, such as the Make Agents Yourself (MAY) framework [Noe12], used to automatically generate agent architecture implementations.

However, as for most general engineering methods, a current limitation of ADELFE is that it only provides high-level guidelines concerning the behavior and architecture of the agents, staying at a general, abstract level. This current limitation makes difficult for a non-expert in AMAS to actually provide an adequate instantiation for the problem he wants to solve. It is the same analysis in [Kad11] which led the author to prone a specialized variant of the method containing additional guidelines and tools for applying AMAS in the context of problem solving.

We will go into greater details concerning the inner workings of ADELFE and how this method was involved in the context of this thesis in chapter 8.

4.2.5 Conclusion on the Adaptive Multi-Agent Systems Theory

We presented here the AMAS theory. This theory proposes a way to model systems by their constituting parts, the interactions between themselves as well as with the environment, and identify the special category of internal cooperative medium systems.

An interesting aspect of this theory is that it provides a guidance to build a multi-agent system based on the problem to solve. Classical solving methods often use a very rigid formalism which needs to be followed. For example, genetic algorithms are a very powerful technique, but they require the problem to fit the genetic representation/fitness function model. To use these kinds of methods, one would now be presented with a whole new problem: “how can I express my problem to fit the solution I want to employ?”

While a non-negligible part of real-world problems are more or less straightforwardly translatable in such formalisms, there is still a whole range of problems for which this translation is not so easy. This can be either because no method adequate enough for the domain was proposed, or because there is no consensual representation of the domain. For these problems, the AMAS theory can provide an interesting asset as the design of the MAS is based on the problem domain. The solution is adapted to the problem, instead of requiring the problem to be adapted to the solution.

We can say that, while the AMAS approach can be applied to any kind of problem, it seems to be especially adequate when trying to solve problems that are still in an exploratory phase, where no “clear-cut” solution exists.

A Multi-Agent System for Continuous Optimization

In the previous part we discussed an inherent limitation of the current continuous optimization approaches. Not only are the current methods highly specialized, but they are also limited by the size, and ultimately by the complexity of the problem. The most complex problems require specific approaches (MDO methods) in order to distribute this complexity while keeping a coherent view. However, these approaches are often complicated to put in practice and tend themselves to be specialized to certain problem types.

In this part we present the main contribution of this thesis: a novel approach to solve complex continuous optimization problems using a MAS. We start by creating a new modeling of continuous optimization problems as entities graphs, and agentify these entities to produce a MAS. We then design agent behaviors and specific mechanisms for the MAS to be able to solve the optimization problem in a distributed way, while maintaining a coherent view of the problem by propagating messages from neighbors to neighbors.

First we propose a new way to model a continuous optimization as a MAS. We want our modeling must be general enough to allow any continuous optimization problem to be transformed in such a way. We also want it simple enough to be automated, which is not only a requirement in the case of big problems, but also a way for the MAS to work "behind the scenes", without requiring the user of the system to have a specific knowledge of multi-agent related concepts. Since optimization specialists often possess expert knowledge and techniques associated to the problems they want to solve, we would like our modeling to be able to integrate with external optimization tools. Our last requirement is to allow the persons in charge of the optimization process to be able to express the problem in the formulation which is the most natural for them, without requiring the problem to be reformulated in any way.

After presenting our modeling, we design an agent-based algorithm for the solving of complex continuous optimization problems. We expose the basic, nominal workflow of the system, how the agents try to optimize the different parts of the system and exchange inform and request messages in order to propagate changes and coordinate the optimization process. We then see how specific configurations related to complex continuous optimization problems can cause this nominal optimization workflow to fail. Using the AMAS theory, we categorize these configurations into different non cooperative situations. For each of these NCSs we propose additional cooperative mechanisms in order to detect, solve the NCS and re-establish the correct optimization process.

At last, we see how our agent modeling can be extended, by proposing some modifications in our system to take in account the handling of uncertainties during the optimization process. We first see how the expression of the problem is changed by the integration of uncertainties. We modify the structure of the information exchanged by the agents and identify some key operations they require in order to be able to manipulate exchanged data. Finally we propose a generic mechanism for experts to be able to express how different representations of uncertainties can be propagated in the context of the problem to solve.

5 Agent-Based Modeling and Simulation of a Continuous Optimization Problem

5.1 NDMO: A Natural Domain Modeling for Optimization

As we previously stated, when solving complex continuous problems existing techniques (*i.e.* MDO methods) usually require a transformation of the initial formulation, in order to satisfy some requirements for the technique to be applied. Beside the fact that correctly applying these changes can be a demanding task for the designers, imposing such modifications changes the problem beyond its original, natural meaning. What we propose here is an agent-based modeling where the original structure, the original meaning of the problem is preserved, because it represents the formulation which is the most natural and easiest for the expert to manipulate. This modeling decomposes the elements of the problem into a graph of entities, which can then be instantiated as agents. We call this modeling *Natural Domain Modeling for Optimization* (NDMO).

To illustrate how an optimization problem is modeled with NDMO, we use the example

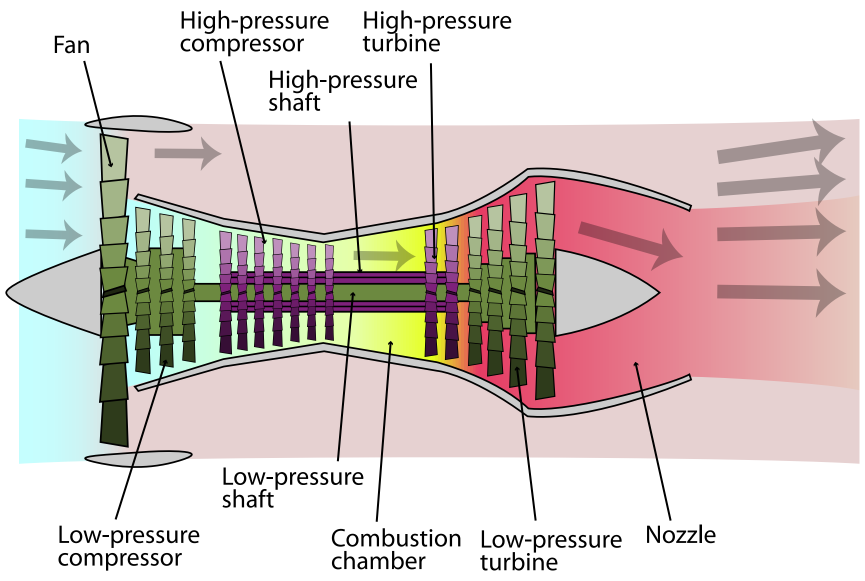


Figure 5.1: Illustration of a Turbofan engine (CC SA-BY K. Aainsqatsi).

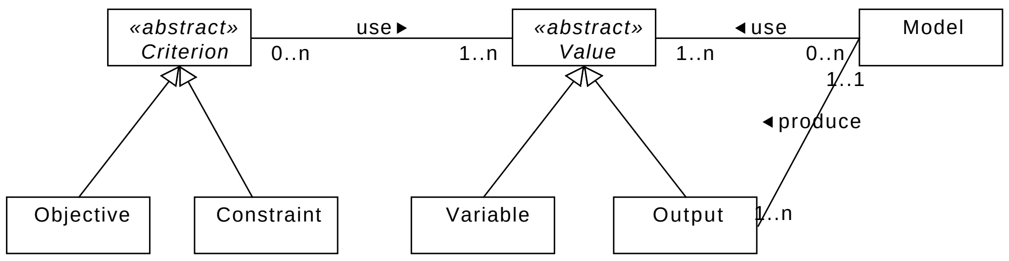
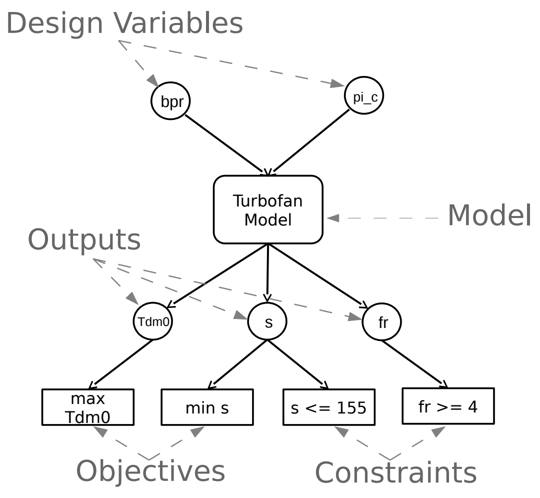


Figure 5.2: Class diagram of MDO problems.

$$\begin{gathered}(Tdm0, s, fr) = Turbofan(pi\_c, bpr) \\ max\ Tdm0 \\ min\ s \\ subject\ to \\ s \leq 155 \\ fr \geq 4\end{gathered}$$

(a) mathematical formulation.



(b) corresponding entities graph.

Figure 5.3: Turbofan problem.

of a simplified turbofan optimization problem. On Figure 5.1, an illustration of the principle of the turbofan can be seen. On such engine, we call bypass ratio the ratio between the air drawn in by the fan not entering engine core (which is *bypassed*) and the air effectively used for the combustion process. We also call pressure ratio the ratio between pressure produced by the compressors and the pressure it receives from the environment.

In order to identify the elements of a generic continuous optimization model, we worked with experts from several related fields: numerical optimization, mechanics as well as aeronautics and engine engineers. As a result, we identified five classes of interacting entities: *models*, *design variables*, *outputs*, *constraints* and *objectives*. These entities and their relations are represented by the diagram in Figure 5.2, that we detail next.

On Figure 5.3a, the analytic expression of this optimization problem is given, while on Figure 5.3b, the problem is presented as a graph of the different entities. The design variables of this problem are pi_c and bpr, which indicate respectively the compressor pressure ratio and the bypass ratio of the engine. The turbofan model produces three outputs: $Tdm0$, s and fr, representing respectively the thrust, fuel consumption and thrust ratio of the engine. In this problem we try to maximize the thrust and minimize the fuel consumption while satisfying some feasibility constraints.

Let us now see in more details the roles of each of these fives entities: *model*, *variable*, *output*, *constraint* and *objective*.

5.1.1 Models

In the most general case, a *model* can be seen as a black box which takes input values (which can be *design variables* or *outputs*) and produces some output values. A *model* represents a technical knowledge of the relations between different parts of a problem and can be as simple as a linear function or a much more complex algorithm requiring several hours of calculation. Often some properties are known (or can be deduced) about a model and specialized optimization techniques can exploit this information. It is important to understand that what exactly is a "model" is quite an arbitrary choice, based on the application domain of the problem. Depending on the goals and needs, a same problem will be divided into models of different granularities and scopes. A model can represent a simple calculus step as well as an entire discipline.

In our Turbofan example on Figure 5.3, the $Turbofan$ function is a *model* entity which calculates the three outputs using the values of *bpr* and *pi_c* (marked π_c in the equations of the model). The equations contained in the model itself are described on algorithm 5.1.

5.1.2 Design Variables

These are the inputs of the problem and can be adjusted freely (within their defined boundaries). The goal of the optimization process is to find the set(s) of values for these variables that maximize the objectives while satisfying the constraints. A *design variables* can be used by *models* to calculate their output values and by *constraints* and *objectives* to calculate their current value. A *design variable* can be shared among several *models*, *objectives* and *constraints*.

Keeping with our example on Figure 5.3, the bypass ratio *bpr* and the compressor pressure ratio *pi_c* are the two *design variables* of our optimization problem.

5.1.3 Outputs

These values are produced by a *model*, and consequently cannot be changed freely. As for the *design variables*, the *outputs* can be used by *models* to calculate their output values and by and by *constraints* and *objectives* to calculate their current value.

In our example on Figure 5.3, the thrust $Tdm0$, the fuel consumption s and the thrust ratio fr are *outputs* produced by the $Turbofan$ model.

5.1.4 Constraints

They are strict restrictions on some parts of the problem, represented as functional constraints defined by equalities and/or inequalities. They can be the expression of a physical limitation, or a requirement concerning the problem.

Regarding the Turbofan on Figure 5.3, the two *constraints* are $s <= 155$ and $fr >= 4$.

Algorithm 5.1: Turbofan Model

```
**Input**: $bpr, \pi_c$

// constants
$\gamma = 1.4$
$cp = 1004.5$
$hfuel = 42.8 \times 1e + 06$
$\alpha_{thrust} = 1.0$
$t0 = 216.7$
$m0 = 0.83$
$\eta_c = 0.9$
$\eta_f = 0.9$
$ttburn = 1560.0$
$\pi_f = 1.7$

$r \leftarrow (\gamma - 1) \times \frac{cp}{\gamma}$
$a0 \leftarrow \sqrt{\gamma \times r \times t0}$
$u0 \leftarrow m0 \times a0$
$\tau_r \leftarrow 1 + \frac{\gamma - 1}{2} \times m0^2$
$\tau_\lambda \leftarrow \frac{ttburn}{t0}$
$\tau_c \leftarrow \pi_c^{\frac{\gamma - 1}{\eta_c \times \gamma}}$
$\tau_f \leftarrow \pi_f^{\frac{\gamma - 1}{\eta_f \times \gamma}}$
$\tau_t \leftarrow 1 - \frac{tau_r}{tau_\lambda} \times ((\tau_c - 1) + bpr \times (\tau_f - 1))$
$UeU0 \leftarrow \sqrt{\frac{\tau_r \times tau_c \times \tau_t - 1}{\tau_r - 1} \times \frac{\lambda}{\tau_r \times \tau_c}}$
$UfU0 \leftarrow \sqrt{\frac{\tau_r \times \tau_f - 1}{\tau_r - 1}}$

// outputs computation
$Tdm0 \leftarrow u0 \times \frac{1}{(1 + bpr)} \times (UeU0 - 1) + \frac{bpr}{1 + bpr} \times (UfU0 - 1)) \times \alpha_{thrust};$
$s \leftarrow \frac{f}{(1 + bpr) \times Tdm0} \times 10e + 06$
$fr \leftarrow \frac{UeU0 - 1}{UfU0 - 1}$
```

5.1.5 Objectives

The goals to be optimized. In the general case, different *objectives* are often contradictory.

The two *objectives* of the Turbofan problems on Figure 5.3 are to maximize th thrust $Tdm0$ and to minimize the fuel consumption s.

Constraints and *objectives* are usually regrouped under the more general term of optimization criteria.

An interesting and important point is that both models, constraints and objectives involve computation. Often the most heavyweight calculus is encapsulated inside a model and the calculi concerning criteria tend to be simple equations, but this is neither an absolute requirement nor a discriminating characteristic.

Using this decomposition, the optimization problem can then be represented as a graph $G = (V, E)$, where the vertices set V is a tuple $\langle \mathcal{V}_d, \mathcal{V}_o, \mathcal{M}, \mathcal{O}, \mathcal{C} \rangle$ (denoting respectively the sets of design variables, outputs, models, objectives and constraints of the problem), and E is the set of relations between the entities. It can be noted that the graph is bipartite regarding the two sets $(\mathcal{V}_d \cup \mathcal{V}_o)$ and $(\mathcal{M} \cup \mathcal{O} \cup \mathcal{C})$, as design and output variables can only be connected to models, objectives or constraints, which can themselves only be connected to design or output variables. Using our Turbofan example, the tuple $\langle \mathcal{V}_d, \mathcal{V}_o, \mathcal{M}, \mathcal{O}, \mathcal{C} \rangle$ of the graph representing this problem is defined as:

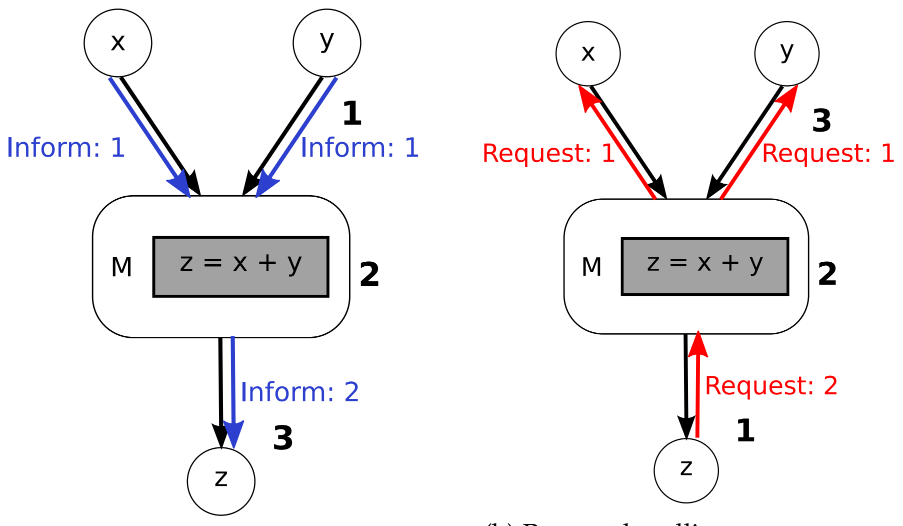
$$\begin{aligned}
\mathcal{V}_d &= \{bpr, pi_c\}, \\
\mathcal{V}_o &= \{Tdm0, s, fr\}, \\
\mathcal{M} &= \{Turbofan\_Model\}, \\
\mathcal{O} &= \{\max Tdm0, \min s\}, \\
\mathcal{C} &= \{s <= 155, fr >= 4\}.
\end{aligned}$$

The NDMO modeling aims to provide the most complete and natural representation of the problem. This modeling preserves the relations between the domain entities and is completely independent of the solving process. The distinct entities that are extracted from the analytical formulation of the problem do not come from an arbitrary decision but are inherent to the problem formulation. On the other hand, all the relationships present between these entities are preserved without any simplification regarding the initial complexity of the problem.

5.2 From an Optimization Problem to a Multi-Agent System

Based on the NDMO modeling in section 5.1, we propose a MAS where each domain entity is associated with an agent. Thus the MAS is the representation of the problem to be solved, with the relationships between agents reflecting the natural structure of the problem. It is worth underlining the fact that this transformation (*i.e.* the agentification) is completely automatic, as it is fully derived from the expression of the problem.

The solving process —constituted by the collective behavior of the agents— basically



(a) Inform handling:
1. receive informs.
2. recalculate outputs.
3. propagate informs.

(b) Request handling:
1. receive request.
2. estimate inputs changes.
3. propagate request.

Figure 5.4: Model agent behavior.

relies on change-value requests sent by the criteria agents, resulting in cooperatively decided adjustments done by the *design variables*. These adjustments lead to new values computed by the models, resulting on the satisfaction or dissatisfaction of the criteria agents. In the same way we presented the different elements of NDMO, we now detail the behaviors of our five agent types: *model*, *variable*, *output*, *constraint* and *objective* agents.

5.2.1 Model Agent

A *model agent* takes charge of a model of the problem. It interacts with the agents handling its inputs (which can be *variable* or *output agents*) and the *output agents* handling its outputs. Its individual goal is to maintain the consistency between its inputs and its outputs. To this end, when it receives a message from one of its inputs informing it of a value change, a *model agent* recalculates the outputs values of its model and informs its *output agents* of their new value (as seen on Figure 5.4a). On the other part, when a *model agent* receives a request from one of its *output agents* it translates and transmits the request to its inputs (as seen on Figure 5.4b).

To find the input values corresponding to a specific desired output value, the *model agent* uses an external optimizer. This optimizer is provided by the engineer based on domain-dependent expert knowledge regarding the structure of the model itself. It is important to underline that the optimizer is used only to solve the local problem of the *model agent*, and is not used to solve the optimization problem globally.

5.2.2 Variable Agent

This agent represents a *design variable* of the problem. Its individual goal is to find a value which is the best equilibrium among all the requests it receives (from models and criteria for

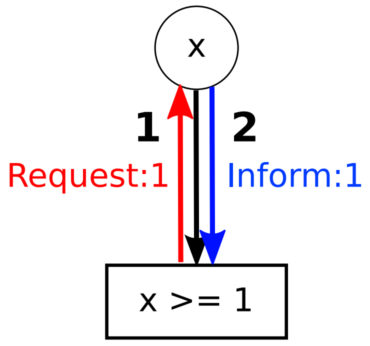
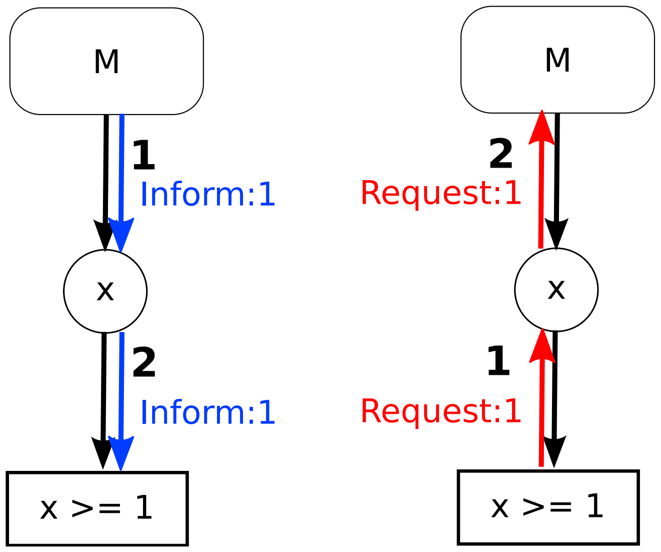


Figure 5.5: Variable agent behavior:
1. receive request.
2. answer with inform.



(a) Inform handling:
1. receive inform.
2. propagate inform.

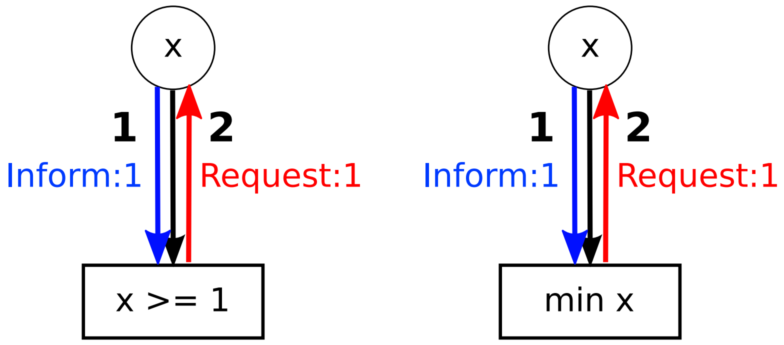
(b) Request handling:
1. receive request.
2. propagate request.

Figure 5.6: Output agent behavior.

which it is an input). The agents using the variable as input can send requests asking it to change its value. When changing value, the agent informs all the related agents of its new value. This behavior is shown on Figure 5.5.

5.2.3 Output Agent

The *output agent* takes charge of an output of a model. *Output agent* and *variable agents* have similar roles, except that *output agents* cannot directly change their value. Instead they send a request to the *model agent* they depend on. In this regard, the *output agent* act as a filter for its *model agent*, selecting among the different requests the ones it transmits. This behavior is summarized on Figure 5.6.



(a) Constraint agent behavior:
1. receive inform.
2. answer with request.

(b) Objective agent behavior:
1. receive inform.
2. answer with request.

Figure 5.7: Constraint and objective agents behavior.

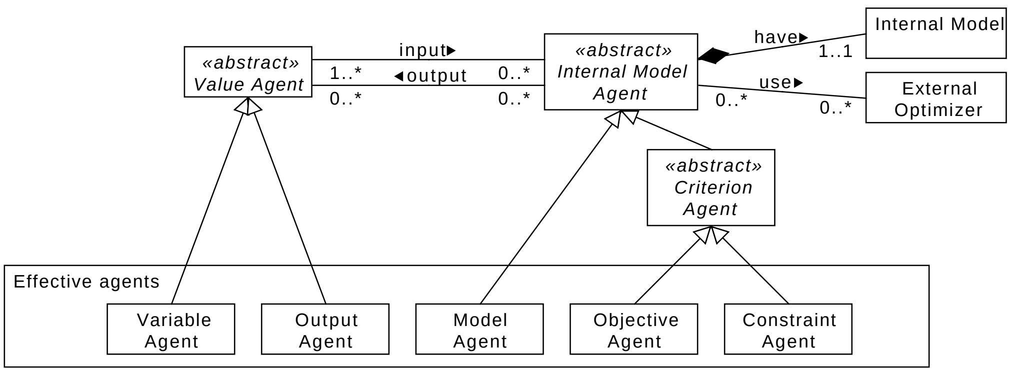


Figure 5.8: MAS class diagram.

5.2.4 Constraint Agent

The constraint agent has the responsibility for handling a constraint of the problem. When receiving a message from one of its inputs, the agent recalculates its constraint and checks its satisfaction. If the constraint is not satisfied, the agent sends *change value* requests to its inputs. The behavior of the agent is illustrated on Figure 5.7a.

It should be noted that, to estimate the input values required to satisfy the constraint on its computed value, this agent employs the same technique as the *model agent* (*i.e.* an external optimizer).

5.2.5 Objective Agent

The *objective agent* is in charge of an objective of the problem. This agent sends requests to its inputs aiming to improve its objective, and recalculates the objective when receiving *value changed* messages from its inputs.

This agent uses an external optimizer to estimate input values that would improve the objective, in the same way than the *model* and *constraint agents*.

This agent modeling is the most direct and inclusive, considering every element of the problem as an autonomous agent. On Figure 5.8 we represent a class diagram of the agents, which is a slightly modified version of the domain class diagram of Figure 5.2. This modified class diagram puts more clearly in light what are the effective agents and how the agents are related with each others by input/output relationships. The bipartite nature of the graph in regard of *value agents* and *internal model agents* is clearly apparent. Two additional, non-agent entities, the *internal model* and the *external optimizer* also made an appearance. The respective roles of these two entities will be detailed in the next chapters.

6 Agents Behavior

We will now discuss the behavior of the five agent types identified in the previous chapter. The functioning of the system can be divided into two main tasks: problem simulation and collective solving.

Problem simulation can be seen as the equivalent of the analysis of classical MDO methods. The agents behavioral rules related to problem simulation concern the propagation of the values of design variables to the models and criteria. For this part, the agents will exchange *inform* messages that contain calculated values. The "messages flow" is top-down: the initial inform messages will be emitted by the variable agents and will be propagated down to the criteria agents. An illustration of the simulation messages flow is shown on Figure 6.1a.

Collective solving concerns the optimization of the problem. The agent behavioral rules related to collective solving are about satisfying the constraints while improving the objectives. For this part, the agents will exchange *request* messages which contain desired variations of values. The "messages flow" is bottom-up: the initial request messages will be emitted by the criteria agents and propagated up to variable agents. An illustration of the solving messages flow is shown on Figure 6.1b.

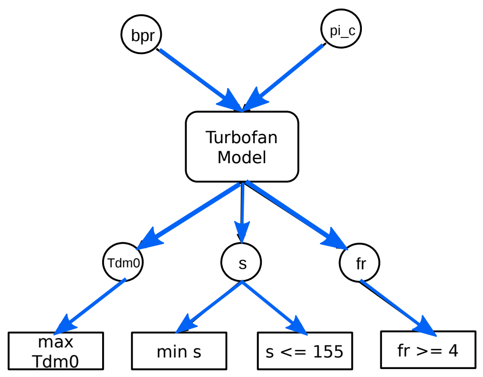
The agents behaviors regarding these two parts can be studied independently, we will thus present them separately. It is however important to remember that these two parts are executed simultaneously. At runtime the agents will simulate the problem and solve it in parallel. Moreover, the different parts of the system will not necessarily work in a synchronous fashion. The effective messages flow of the system will more probably be akin to the Figure 6.1c.

6.1 Problem Simulation

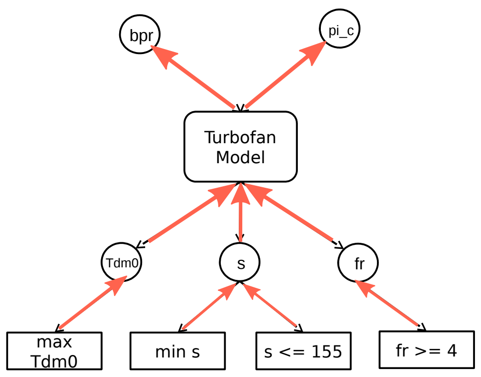
In this section we will present the agents behaviors related to the simulation of the problem. Regarding this part, the main concern of the agents is to ensure consistency between the values of the design variables and the produced outputs. To this end, the agents will propagate *Inform* messages through the system.

An inform message carries a new value v. The exact semantic of this information slightly changes depending on which agents are involved:

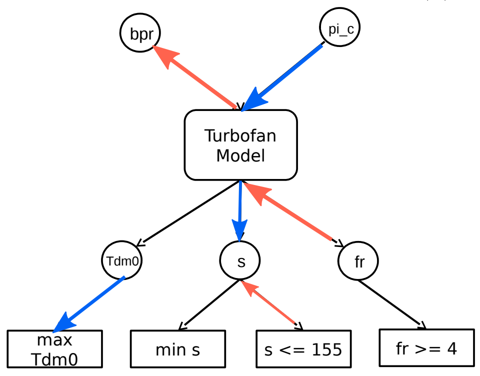
▷ If the message is sent from a value agent (variable or output) to a model or criterion agent, it indicates to the receiving agent that the sending agent has changed of value.



(a) Informs flow for simulation.



(b) Requests flow for solving.



(c) Effective messages flow.

Figure 6.1: Messages flow for simulation and solving.

- ▷ If the message is sent from a model agent to an output agent, it indicates to the receiving agent that the model has calculated its new value.

In practice this distinction is not fundamentally important to understand the functioning of the system.

As stated in section 5.1, models, constraints and objectives involve a specific calculation operation. For example the constraint $x - y \geq 0$ implies the very basic calculation $x - y$. We regroup these operations under the term *internal models*. As we said, no specific hypothesis is done concerning the nature of an internal model and, more importantly, no distinction is done regarding the internal model used by a model agent and the one used by an objective or a constraint agent.
While in some cases specific informations can be known regarding the nature of an internal model, in the most general case these internal models can be seen as black boxes and are handled as such by the agents.

6.1.1 Variable Agent

Regarding problem simulation, the role of the variable agent is to ensure that the agents to which it is connected know its current value. A variable agent has to send new Inform messages when:

- ▷ new agents are connected to it (typically at the creation of the system).
- ▷ it decides to change its value based on received requests (see 6.2.1).

▷ the designer changed its value.

The behavior of the variable agent is summarized in algorithm 6.1. First of all the variable checks if it changed its value (typically as the consequence of receiving requests during the solving phase). If it is the case it sends its new value to its contacts. It then checks if new agents have requested to be added to its contacts. If it is the case, it adds them and sends them its current value, whether its value changed or not, in order for the new contacts to be informed of the current value of the variable.

Algorithm 6.1: Problem Simulation – Variable Agent Behavior

```
$C \leftarrow$ previously connected agents
$C' \leftarrow$ newly connected agents
$v \leftarrow$ previous value
$v' \leftarrow$ new value
// notify contacts of value change
if $v \neq v'$ then
    $v \leftarrow v'$
    foreach agent $a \in C$ do
        send($a$, new Inform($v$))
    end
end
// send its value to new contacts
// even if it did not change
if $C' \neq \emptyset$ then
    foreach agent $a \in C'$ do
        send($a$, new Inform($v$))
    end
    $C \leftarrow C \bigcup C'$// memorize new connected agents
end
```

6.1.2 Model Agent

A model agent must maintain consistency between its inputs and outputs. That is, it must ensure that the agents handling its output variables are informed of their new values when the inputs changes. When receiving an Inform message from one of the agents controlling its inputs, the model agent reevaluates its internal model, taking the new value into account. Using the new values produced by the internal model, the model agent then sends Inform messages to the agents responsible of its outputs.

All outputs of a model agent are not necessarily associated with an output agent. For example, if the designer is not interested by the value represented by one of the outputs, he is not required to represent it by an agent. In this case the model agent will silently calculate the new value of the output at the same time than the others, but will not propagate it[1].

However, as the problem is dynamic, the designer can decide at any time to connect a new output agent to a previously unconnected output. In this case, the model agent must

[1]Remember that the internal model of the agent is a black box, consequently the agent cannot choose to evaluate it partially. If the evaluation of the output could has been done independently from the others, it may be preferable to create two separate models

send to the output agent the last value it calculated for this output.

The behavior of the model agent is summarized in algorithm 6.2.

Algorithm 6.2: Problem Simulation – Model Agent Behavior

```
$C \leftarrow$ previously connected output agents
$C' \leftarrow$ newly connected output agents
$\{v_o\} \leftarrow$ previous output values
$\{v_i\} \leftarrow$ new input values
if $\{v_i\} \neq \emptyset$ then
    // use internal model to recalculate outputs
    $\{v_o\} \leftarrow Internal\_Model(\{v_i\})$
    foreach agent $a_o \in C$ do
        // inform output agents of their new value
        send($a_o$, new Inform($v_o$))
    end
end
// send their values to new outputs
// even if they did not change
if $C' \neq \emptyset$ then
    foreach agent $a_o \in C'$ do
        send($a_o$, new Inform($v_o$))
    end
    $C \leftarrow C \bigcup C'$// memorize new outputs agents
end
```

6.1.3 Output Agent

For the problem simulation, the output agent has a very similar role to the variable agent. It tries to ensure that the agents to which it is connected know its current value. It has to change new inform messages when:

- ▷ new agents are connected to it.
- ▷ it receives an inform message from its model agent, indicating it its new value.

Unlike the variable agent, the output agent value should not be directly changed by the designer, as it represents a non-free variable.

The behavior of the output agent is summarized in algorithm 6.3.

At runtime, the designer can interact with the MAS. Among the possible interactions, the designer can remove the model calculating an output. In this case the output agent would become a variable agent and change its behavior accordingly. The inverse transformation is also possible, a variable agent could suddenly be “plugged” in an output of a model and would become an output agent.

6.1.4 Constraint/Objective Agent

In regard to problem simulation, constraint and objective agents only role is to update their value based on the Inform they receive from their inputs.

Algorithm 6.3: Problem Simulation – Output Agent Behavior

```
$C \leftarrow$ previously connected output agents
$C' \leftarrow$ newly connected output agents
$v \leftarrow$ previous value
$v' \leftarrow$ new value
// notify output contacts of value change
if $v \neq v'$ then
    $v \leftarrow v'$
    foreach agent $a \in C$ do
        send($a$, new Inform($v$))
    end
end
// send its value to new output contacts,
// even if it did not change
if $C' \neq \emptyset$ then
    foreach agent $a \in C'$ do
        send($a$, new Inform($v$))
    end
    $C \leftarrow C \bigcup C'$// memorize new connected agents
end
```

Their behavior is summarized in algorithm 6.4.

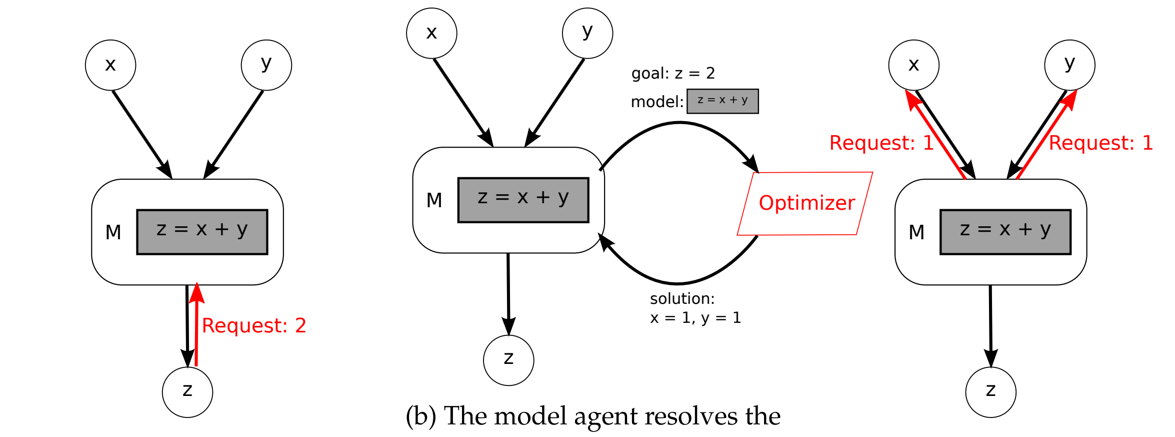
Algorithm 6.4: Problem Simulation – Constraint/Objective Agent Behavior

```
$\{v_i\} \leftarrow$ new input values
if $\{v_i\} \neq \emptyset$ then
    // use internal model to recalculate outputs
    $\{v'_o\} \leftarrow Internal\_Model(\{v_i\})$
end
```

6.2 Collective Solving

During solving, the criteria agents try to improve their local goals. That is, the constraint agents try to keep their constraint satisfied, while the objective agents try to improve their objective. To this end, they send *Request* messages to the agents controlling their inputs, asking them to change value. The others agents have to propagate these requests toward the variable agents in the most adequate way.

However a specific difficulty arises. As explained in the previous section, model, constraint and objective agents manipulate the underlying equations (or algorithms, or responses surfaces *etc.*) of the problem through internal models, which are in most cases black boxes. In this case, how can constraint agents know which values to ask to satisfy their constraint? How can objective agents know the values that improve their objective? How can model agents know which values for their inputs could produce the values required by their outputs?
For the internal model agents to be able to work, the designer must supply them with an *external optimizer*. The role of this optimizer is the following: if we see an internal model



(a) The model agent receives a request.

(b) The model agent resolves the request using an external optimizer.

(c) The model agent transmits the new requests to its inputs.

Figure 6.2: Use of an external optimizer.

as a function[2] providing an output O from a set of input I, what we need is to be able to estimate the inverse function, providing the values of I corresponding to a specific O. This specific problem can easily be expressed as a classical optimization problem in itself, which can be solved by classical optimization methods. As the agents manipulate black boxes, it is (nearly[3]) impossible for them to do this estimation by themselves. The designer, which has a more advanced knowledge of the internal models, can provide an adequate optimization method for each of them. In the most general case it can be assumed that each agent in charge of an internal model uses its own specific optimizer. However in practice several instances of the same optimizer can be used by different agents, if the designer deems that their models present sufficient similar characteristics for the optimization method to be efficient for each of them. An example of use of an external optimizer by a model agent can be seen on Figure 6.2.

As a remark, finding the inputs corresponding to a desired value is not directly an optimization problem, but is trivial to translate into one. For example, the problem can be formulated as finding I that minimizes $(f(I) - O)^2$. Alternatively, the objective-function can be replaced by a constraint such as $f(I) = O$. We do not impose a specific transformation method, letting the expert which implements the optimizer chooses which transformation is the most efficient in regard to the specificities of the optimization method.

With this specificity in mind, let us now examine the behavior of the different agent types in the context of collective solving.

6.2.1 Variable Agent

During solving, a variable agent is susceptible to receive change requests from other agents. When a variable agent receives a change request, it tries to change its own value in order to accommodate the requester while taking into account the previous demands of the rest of its neighbors. To this end, the variable agent uses an exploration strategy based on

[2]Calling an internal model a function is a simplification. Since it is a black box it may have an internal state. Consequently the same inputs may not always produce the same outputs

[3]We will see in section 6.2.2 how the agents can extract enough information at runtime to provide their own basic optimizer

Adaptive Value Tracker (AVT) [LCG11]. The AVT is an adaptation of dichotomous search for dynamic values. The idea is to change value according to the direction which is requested and the direction of the past requests. While the value varies in the same direction, the variation delta is increased so the value varies more and more. As soon as the requested variation changes, it means that the variable went past the good value, so the variation delta is reduced. This capability to take into account a changing solution allows the variable agent to continuously search for an unknown dynamic target value. This capability is required for the system to be able to adapt to changes made by the engineer during the solving process.

The algorithm of the AVT is the following. Given the following variables:

- ▷ $v_t \in [v_{min}; v_{max}]$ the current value of the AVT.
- ▷ $\Delta_t \in [\Delta_{min}; \Delta_{max}]$ the last variation of v_t.
- ▷ λ_i the increase coefficient of the AVT ($\lambda_i > 1$).
- ▷ λ_d the decrease coefficient of the AVT ($0 < \lambda_d < 1$).

When receiving a new adjustment feedback $Fb_t \in \{\uparrow; \downarrow; \sim\}$ (corresponding respectively to "increase current value", "decrease current value" and "keep current value"), a new value v_{t+1} is calculated using Table 6.1.

Table 6.1: AVT behavior

		Fb_t		
		$\uparrow$	$\downarrow$	$\sim$
Fb_{t-1}	$\uparrow$	$\Delta_t = \Delta_{t-1} \times \lambda_i$ $v_{t+1} = v_t + \Delta_t$	$\Delta_t = \Delta_{t-1} \times \lambda_d$ $v_{t+1} = v_t - \Delta_t$	$\Delta_t = \Delta_{t-1} \times \lambda_d$ $v_{t+1} = v_t$
	$\downarrow$	$\Delta_t = \Delta_{t-1} \times \lambda_d$ $v_{t+1} = v_t + \Delta_t$	$\Delta_t = \Delta_{t-1} \times \lambda_i$ $v_{t+1} = v_t - \Delta_t$	$\Delta_t = \Delta_{t-1} \times \lambda_d$ $v_{t+1} = v_t$
	$\sim$	$\Delta_t = \Delta_{t-1}$ $v_{t+1} = v_t + \Delta_t$	$\Delta_t = \Delta_{t-1}$ $v_{t+1} = v_t - \Delta_t$	$\Delta_t = \Delta_{t-1} \times \lambda_d$ $v_{t+1} = v_t$

In our experiments we kept the default proposed values of $\lambda_i = 2$ and $\lambda_d = \frac{1}{3}$.

One could wonder why the variable agent uses such a seemingly contrived method instead of just taking the requested value. It should be remembered that the variable agent can be connected to several parts of the problem and can receive contradictory requests from different origins over time (as we said, the case where the variable receives contradictory requests at the same time will also be studied later). Thus, the variable agent must have a way to take into account these contradictory requirements to find an adequate compromise. Moreover, until the system (or at least this part of it) has stabilized, the values requested has very little chance of being exact. The value can thus be considered as an indication of the direction in which the requesting agent desire the variable to go. As the system stabilizes, the precision of the AVT is improved and the value of the variable is closer and closer to the exact correct value.

While changing value based not on the value requested but on the direction can seem paradoxical, it is necessary. Since all agents have only a local view of the system (itself plus its neighbors) the requests they make are often approximate. Consequently they need to iterate several times. If the search space is large, the system could take a long time to converge toward the solution. By using a near-dichotomous strategy, we greatly accelerate this convergence.

This behavior is summarized in algorithm 6.5.

Algorithm 6.5: Collective Solving – Variable Agent Behavior

```
$v \leftarrow$ old value
$v_r \leftarrow$ requested value
$avt \leftarrow$ Adaptive Value Tracker module
if $v < v_r$ then
    $feedback \leftarrow$ INCREASE
else
    $feedback \leftarrow$ DECREASE
end
$v' \leftarrow avt$.adjustValue($feedback$)
```

6.2.2 Model Agent

The model agent is responsible for transmitting requests it receives from its outputs to its inputs. To be able to translate the ingoing requests to its inputs, the model agent uses an external optimizer, as presented at the start of the section. When receiving a request from one of its outputs, the model agent calls the external optimizer, which provides it with the adequate input values corresponding to the request. The model agent then sends requests corresponding to these values to its inputs. While the model agent can seem to be very simplistic, we will see in section 6.3 how its behavior can be expanded to handle multiple problematic cases.

The behavior of the model agent is summarized in algorithm 6.6.

Algorithm 6.6: Collective Solving – Model Agent Behavior

```
$v_r \leftarrow$ requested value from output
$M \leftarrow$ the internal model of the agent
$O \leftarrow$ associated optimizer
// get the corresponding input values
// estimated by the optimizer
$\{v_r^i\} \leftarrow$O($v_r$, $M$)
foreach $i \in$ input controlling agents do
    // send the requests corresponding to the estimated values
    send($i$, new Request($v_r^i$))
end
```

Using an Internal Optimization Algorithm as an Alternative to External Optimizers

At the start of the section we stated that, since its internal model is a black box, the model agent cannot in itself perform the “bottom-up” translation from the values requested by its outputs to values for its inputs, requiring an external optimizer to carry such task. While this is true at the start of the process, the model agent can at runtime observe the values returned by the internal model and make some estimations about its internal topology. We present now such a mechanism which can be integrated into model agents, capable of taking advantage of the functioning of the agent using simple black box optimization techniques.

This mechanism integrates itself during the simulation behavior of the agent. Remember that, as presented in section 6.1.2, when receiving a new value from one of its inputs the model agent calls its internal model to recalculate the values of its outputs. At this step, the agent can observe the impact of the variation of the input on each output. We call *correlation* between the input i and output o the variation of o relative to the variation of i, calculated as $\frac{v_o^{t+1} - v_o^t}{v_i^{t+1} - v_i^t}$. This correlation can be seen as a local linear approximation of the partial derivative $\frac{\delta o}{\delta i}$. This correlation can then be used during solving to estimate the required changes from the inputs to satisfy a change requested by an output.
For example, suppose a model with one input i and one output o. During simulation, the value of i changes from 1 to 2. As a consequence the internal model provides a change of o from -2 to -4. Thus the correlation between i and o is $\frac{-4-(-2)}{2-1} = -2$. The output o then sends a request to the model agent to take the value 4. The agent can estimate that for o to take the value 4, i needs to take the value $\frac{4}{-2} = -2$. The complete formula when several inputs are involved in the calculus of o is presented with the summary of the mechanism in algorithm 6.7.

We presented here a very simple learning mechanism for the agent to default when it is not provided with an external optimizer. This mechanism is voluntarily simple and lightweight, in order to be applicable to a broad range of model topologies. While it is possible to make further refinements to improve its results, we believe that such needs are better covered by the use of external optimizers provided by experts of the problem application domain, which can be tailored for specific needs.
Interestingly, our approximation technique can also be used to create local linear approximations of complex black box models. We will see in the later sections how such information can be used by the system to solve some corner cases.

6.2.3 Output Agent

Output agents have a more "passive" role than variable agents during problem solving. As it depends on a model agent for its value, an output agent will simply transmit the requests it receives to its model, as summarized in algorithm 6.8.

6.2.4 Constraint Agent

With the objective agents, the constraint agents are the origins of the requests which are propagated into the system. The goal of a constraint agent is to ensure its constraint is satisfied. To this end, the constraint agent calculates a target value for the constraint, estimates corresponding target values for its inputs and sends them as requests to the agents in charge of these inputs. When the constraint is satisfied, the constraint agent tries to ensure

Algorithm 6.7: Collective Solving – Internal Optimizer Algorithm

```
// estimating correlation deltas
$I \leftarrow$ current input values
$O \leftarrow$ current output values
foreach $inform \in$ {received inform messages} do
    $input \leftarrow$ the input concerned by $inform$
    $i_{t-1} \leftarrow$ old value of $input$
    $i_t \leftarrow$ new value of $input$
    $\{o_{t-1}\} \leftarrow \{$ old value of $o, \forall o \in O\}$
    // reevaluate the outputs using the new value of the inform
    // (the others inputs are not updated)
    $I_t \leftarrow \{i \in I, i \neq i_{t-1}\} \cup \{i_t\}$
    $\{o_t\} \leftarrow M(I_t)$
    // for each output, calculate the new correlation $\delta$
    foreach $o \in O$ do
        $\delta_{i,o} \leftarrow \frac{o_t - o_{t-1}}{i_t - i_{t-1}}$
    end
end

// linear approximation optimization
$o^* \leftarrow$ target value for $o$
$\Delta_o \leftarrow o^* - o_{current}$
$\{\delta_i\} \leftarrow$ estimated correlations between all the inputs and the output
foreach $i \in inputs$ do
    // calculate the input variation
    $\Delta_i \leftarrow \Delta_o \times \frac{\delta_i}{\sum_{i \in \text{inputs}} \delta_i}$
end
```

Algorithm 6.8: Collective Solving – Output Agent Behavior

```
$v_r \leftarrow$ requested value
$m \leftarrow$ the model agent responsible of the output
send($m$, new Request($v_r$))
// forward the request to the model agent
```

it will keep being satisfied. To this end it continues to send requests to move its current value further and further from the constraint threshold.

The problem of finding corresponding input values for its target value is similar to the one of the model agent of finding adequate input values from the outputs. As we said at the end of section 5.1, constraint agents (and objective agents) are similar to model agents in the fact that both have control of an *internal model*. Thus, constraint agents can use an external optimizer to find adequate target values for its inputs. The only difference with model agents being that, instead of having a value requested by an output, the constraint agent estimates itself its own target value.
In the same way, constraint agents can use the same internal optimization mechanism than model agents presented in algorithm 6.7, when not provided with an external optimizer.

The behavior of a constraint agent with a "lower or equal" constraint is described in algorithm 6.9. This algorithm is easily adapted to the others constraint types.

Algorithm 6.9: Collective Solving – Constraint Agent Behavior

```
$v \leftarrow$ current value
$v_t \leftarrow$ threshold value
$\Delta^* \leftarrow |v - v_t|$
$\{v_i^*\} \leftarrow Optimizer(v - \Delta^*)$
// can be either external optimizer or internal optimization
   mechanism
foreach $i \leftarrow inputs$ do
    // change the estimated target value to the input
    send($i$, new Request($v_i^*$))
end
```

It should be noted that, even when the constraint is satisfied, the constraint agent continues to send requests to its inputs. This behavior can be justified by several reasons. First of all, the goal of the constraint agent is to ensure its constraint stays satisfied, so when the constraint is already satisfied the constraint agent has interest in "pushing" its current value afar from the constraint frontier.
The other reason is to help its neighbors agents to keep a coherent view of their relations. Indeed, as the system is interactive, the designer can decide at any time to remove or change the constraint. As a consequence it is not possible for the agents to simply keep in memory the constraints they are linked to, as these informations can become obsolete at any time. A simple way to remedy this situation is for the constraint to not assume that the others agents will memorize its requirements, and to keep sending them requests.

6.2.5 Objective Agent

Along with constraint agents, the objective agents are the origins of the requests which are propagated into the system. The goal of an objective agent is to improve its objective. To this end, the objective agent iteratively estimates a new target value for its objective, finds the corresponding input target values and sends them as requests to the input agents. The behavior of the objective agent in this regard is quite similar to the one of the constraint agent. The difference is that, while the constraint agent has a reference value to target (the threshold value), the objective agent has no such value. Instead, the objective agent will target a new value based on the last variation of its value.

Like the constraint and model agents, the objective agent can use an external optimizer to find the corresponding input values, or can use the internal optimization mechanism presented in algorithm 6.7.

The behavior of an objective agent with a "minimize" objective is described in algorithm 6.10. This algorithm is easily adapted to a maximization objective.

Algorithm 6.10: Collective Solving – Objective Agent Behavior

```
$v \leftarrow$ current value
$\Delta_{t-1} \leftarrow$ last variation of value
$\{v_i^*\} \leftarrow Optimizer(v - \Delta_{t-1})$
// can be either external optimizer or internal optimization
   mechanism
foreach $i \in inputs$ do
    // change the estimated target value to the input
    send($i$, new Request($v_i^*$))
end
```

It is interesting to note that an objective agent exhibits a behavior similar to the one of the constraint agent, but for slightly different reasons. While the constraint agent continuously sends requests in order to make its constraint safer and safer, the objective agent continuously sends requests because it can never know if it reached the best value for its objective. An objective agent is essentially "blind", as the objective is a black box, and must rely on the external optimizer (or the internal optimization mechanism) in order to improve it. Consequently, the objective agent never stops trying to find a better value. As with constraint agents, this mechanism is quite handy in the context of interactive optimization, as the designer can completely change the nature of the objective at any moment.

An important point is that each agent only has a partial knowledge and local strategy. No agent is in charge of the optimization of the system as a whole, or even of a subset of the other agents. Contrary to the classical MDO methods presented earlier, the solving of the problem is not directed by a predefined methodology, but by the structure of the problem itself. The emerging global strategy is unique and adapted to the problem.

6.2.6 Adaptive Agents and Co-design

As previously said, one of our goals is for the system to be able to adapt to changes made in their environment, in order to allow the expert to experiment with the problem and observe the impact on the solution. This capability is especially relevant when the optimization problem is the representation of a complex system to design, as the designer may often have only an imperfect knowledge about the system to be designed. Using our system the designer can change the problem and directly observe the effects on the design.

However, in the behavioral algorithms we presented, we made nearly no mention of

Algorithm 6.11: Agents Behaviors Synthesis

```
behavior of Model Agent
    repeat
        analyze received messages
        if received new information messages then
            recalculate outputs
            inform depending agents
        end
        if received new requests then
            use optimizer to find adequate inputs
            propagate requests to input agents
        end
    until resolution end
behavior of Variable Agent
    repeat
        analyze received messages
        if received new requests then
            adjust value
            inform depending agents
        end
    until resolution end
behavior of Output Agent
    repeat
        analyze received messages
        if received new information messages then
            update its value
            inform depending agents
        end
        if received new requests then
            transmit requests to model agent
        end
    until resolution end
behavior of Constraint/Objective Agent
    repeat
        analyze received messages
        if received new information messages then
            update its value
            use optimizer to find adequate inputs
            send new requests to variable/output agents
        end
    until resolution end
```

specific mechanisms for handling dynamic aspects of the problem. The reason is simple: these mechanisms are already sufficient to take into account such dynamics, as we will demonstrate in the experiments. This intriguing property can be explained by two factors. First of all the agents had to be designed considering that previous information they received can become outdated, as the different parts of the optimization problem do not necessarily converge at the same rate. To this first observation is added the fact that each agent has to keep its reasoning to a local level, since an agent trying to reason on a global level would eventually be overwhelmed by the complexity of the problem.

The consequence of these two factors is that agents indifferently handle changes caused by the "normal" exploration of the search space and changes made to the optimization problem by the designer. Indeed, from the point of view of an agent, a complex problem and a dynamically changing problem are indistinguishable, and mechanisms tailored to handle one will also help solving the other.

Therefore we can say that, by forcing ourselves to keep the agent behavior at a local level, we gained another advantage in addition of the scalability properties of the MAS. Complex optimization problems require the agent reasoning to be kept at a local level, both in space (neighborhood and information perceived by the agents) and in time (memorization of the information). In these conditions, the agent is then "naturally" able to take external changes into account, as they are not distinguishable, from its point of view, from "normal" changes due to the optimization process. This characteristic which could at first sight be perceived as a handicap for the agent is in fact a bearer of several benefits.

6.3 Non-Cooperative Situations

In the previous sections, we presented the basic agents behaviors of our system. Algorithm 6.11 is a synthesis of the behavior of each agent type. While this basic behavior could suffice in very simple test cases, it is not sufficient to handle the specificities of most continuous optimization problem configurations. In these situations, this nominal agent behavior would lead to a suboptimal result.
Based on the AMAS theory (presented in section 4.2), we can consider these configurations to be Non Cooperative Situations (NCSs), as they represent a situation where, because of a shortcoming in the behavior of the agents, the system does not produce an adequate functionality (in our case, the optimization of the problem).
Consequently, we need for each NCS to provide the agents with specific mechanisms to:

1. detect occurring instances of the NCS
2. solve detected instances of the NCS
3. if possible, anticipate future instances of the NCS and avoid them

Methodologically, by studying how the system handles specific problems with characteristics which are specific to continuous optimization (interdependencies, conflicting criteria *etc.*), we identified several problematic configuration types, and defined different cooperation mechanisms for the agents that allow the system to correctly solve problems which exhibit these characteristics.

We will now present the NCSs we identified as well as the solving mechanisms we propose to handle them.

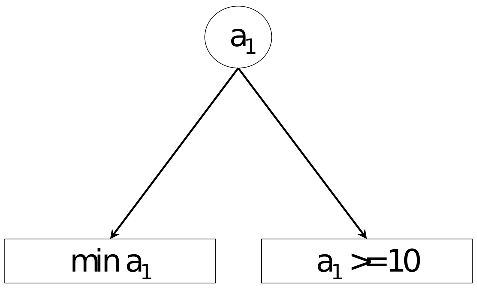


Figure 6.3: Conflicting trajectories example.

6.3.1 Conflicting Requests

The first problematic configuration, and arguably the most obvious, is the one where an agent is in position of receiving simultaneous conflicting requests. This situation concerns every agent which is susceptible to receive requests: variable, model and output agents. This situation can be qualified as a *Conflict* NCS, as the requesting agents ask for contradictory modifications of their environment. A minimal example of configuration in which conflicting requests may appear is shown in Figure 6.3.

When such an agent receives contradictory requests, it needs a way to select which request to apply and which request to reject. We can intuitively see how this decision should be based on the current state of the requesting agents. For example, when receiving requests both a satisfied and a non-satisfied constraint, a variable agent should favor the non-satisfied one. In order for the agents to be able to make this decision, we need to provide them a way to compare the states of different agents.

Solving Conflicting Requests using Criticality To this end, we introduce a new mechanism based on a specific measure called *criticality*. This measure represents the state of dissatisfaction of the agent regarding its local goal. Each agent is in charge of estimating its own criticality and providing it to the other agents[4]. The role of this measure is to aggregate into a single comparable value all the relevant indicators regarding the state of the agent. Having a single indicator of the state of the agent is interesting as it simplifies the reasoning of the agents. In addition, this mechanism has the interesting property of limiting the informations transmitted to the others agents, which can be of interest in case of a distributed optimization where data privacy is an issue. However the system designer has the difficult task to provide the agents which adequate means to calculate their criticality. Also, in specific cases, this information on the state of the agents is not sufficient to take the correct decision, as we will see in the following sections.

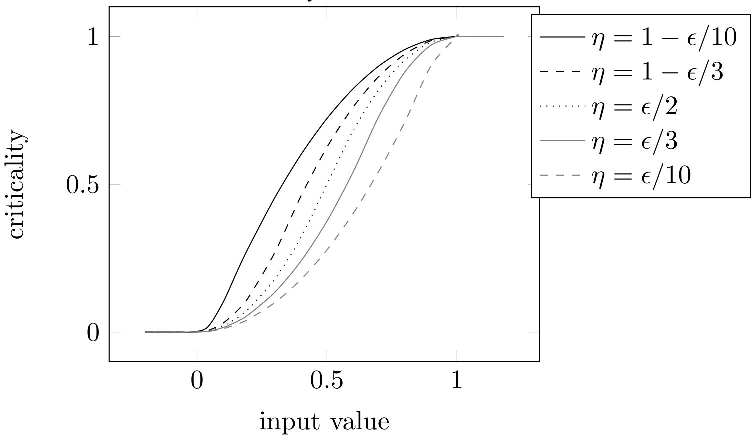
In the proposed system, criticality is initially provided by constraint and objective agents and is propagated in the system through their requests. Let us illustrate this with a constraint of the type $g(X) \leq t$, with X input of the constraint, *g(X)* the constraint equation and t the threshold under which the constraint is satisfied. The basic requirements regarding the criticality of this agent is to be low when the constraint is satisfied and high when the constraint is violated. Thus, the criticality of this agent is function of its current value and of

[4]We do not concern ourselves with the problematic of trust here, each agent is assumed to provide the most trustful and accurate information without cheating or lying. The use of measures such as criticality in open and untrusted environments is in itself an interesting question to say the least.

$$criticality_{t,\eta,\epsilon}(x) = \begin{cases} 0 & \text{if } x < t-\epsilon\,, \\ -\gamma(t-x-\eta)^2/(2(\epsilon-\eta)) + \gamma(t-x-\eta) + \delta & \text{if } t-\epsilon \leq x \leq t-\eta, \\ \gamma(-t-x-\eta)^2/(2\eta) + \gamma(-t-x-\eta) + \delta & \text{if } t-\eta \leq x \leq t, \\ 1 & \text{if } x > t \end{cases}$$

where
$\gamma = -2/\epsilon$,
$\delta = -\gamma(\epsilon - \eta)/2$,
and $0 < \eta < \epsilon$.

(a) Analytical formulation.



(b) Shapes of criticality function of threshold t = 1 for $\epsilon = 1$ and different η.

Figure 6.4: Criticality function of a constraint agent.

the threshold.

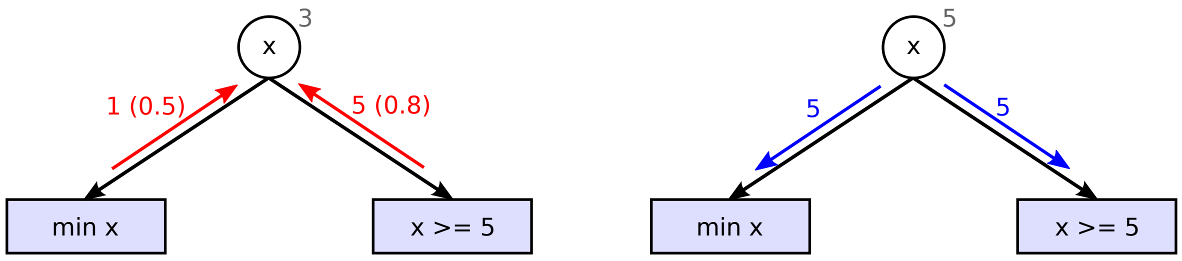
To compute it, we use the barrier function defined on Figure 6.4a. It takes as input x, the current value of the constraint. It is parameterized by t, the threshold, and by η and ϵ that both regulate the shape of the function as seen on Figure 6.4b. Its value always varies between 0 and 1. The ϵ can be adjusted by a domain expert if needed: the higher it is, the faster the constraint increases in criticality. In our experiments, we used $\epsilon = 0.1$ and η was set to roughly a third of ϵ, *i.e.* 0.03. This function allows a smooth transition between two states and provides several interesting properties: it is continuous, differentiable, requires few parameters, is computed quickly and is relatively easy to grasp.

The criticality of the other agents is determined as follow:

- ▷ For objective agents: the criticality is set to an arbitrary constant value which must be lower than 1. In our experiments we settled for a value of 0.5. This translates the fact that, in the general case, an objective could theoretically always be improved, but is less important to satisfy than a constraint.
- ▷ For variable, output and model agents: the criticality is set to the highest criticality among the received requests.

When the system converges to a solution, it stabilizes at a point where the maximum of the criticalities of the agents is minimized.

An illustration of this mechanism can be seen on Figure 6.5. In this example, a variable agent (with a current value of 3) receives contradictories requests from an objective (decrease) and a constraint (increase). As the constraint is not satisfied, the criticality associated to its



(a) The criterions send their requests and current criticalities (in parenthesis).

(b) The variable agent selects the most critical request and responds accordingly.

Figure 6.5: Criticality mechanism (the criticality of the requests is indicated in parenthesis).

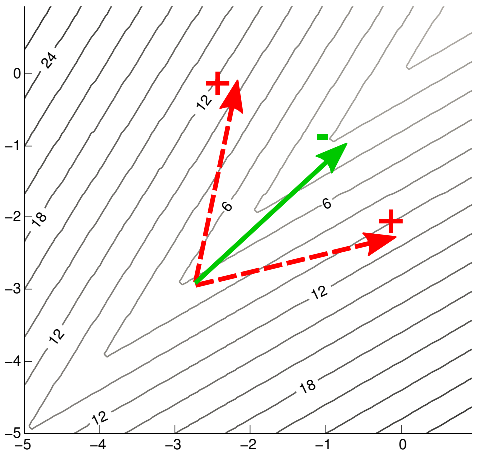


Figure 6.6: Illustration of coordination requirements on a basin function with 3 different trajectories.

request (0.85) is higher than the one of the objective (0.5). As a consequence the variable agent selects the request of the constraint agent and increases its value.

Maintaining a Correct Trajectory with Successive Contradictory Requests Another aspect of the handling of contradictory requests concerns the possibility for the agent to receive successively contradictory requests, possibly from the same origin. This situation can arise from the fact that some problem topologies requires the agents to move in a coordinated way across the search space. This difficulty is illustrated on Figure 6.6. On this figure is presented a 2-inputs functions we want to minimize (top-right corner), and 3 different trajectories. For each of these trajectories, the two inputs vary in the same direction, but with different strengths.
We can see that, for the two dashed trajectories, the objective-function increases as the variation of one of the input is much greater than the variation of the other. Only on the non-dashed trajectory the objective is actually improved. The implication is that, even if all the inputs vary in the correct direction, they can cause a degradation of the objective if they do not coordinate the strengths of their variations.

With the nominal behavior, the agents would iterate and notice their mistake, compensating by changing direction. However this behavior would lead to a very inefficient trajectory. Consequently it is much more preferable that the variable agents coordinate their respective movements in order to maintain a good trajectory. However, these variable are not related to each others in any way, and cannot communicate with each others. While we could create additional links to make the variables communicate among themselves, it would violate our locality principle. Instead, we propose a mechanism which base itself on the fact that the agents do not directly know each others, but know that the others agents exist and are cooperative. Consequently, by making careful adjustments and observing the feedback of their environment, the agents can be able to estimate the impact of their variation in regard of the variations of the others variable agents, and correct their movement in consequence.

The basic idea is for the variable agent to create a representation of the trajectory it is following (in the 1-dimension space). It uses a *trending trajectory* measure, noted tt, whose value varies between -1 and 1. A positive value for tt means that the variable follow an *increase* trajectory, while a negative value implies a *decrease* trajectory. The higher the value of $|tt|$, the stronger the confidence of the agent into the trajectory.

After the variable agent has selected a request, it updates tt based on the direction requested. If the selected request asks the variable agent to decrease, tt is decreased and *vice versa*. The exact impact of the new request on tt is controlled by a trending coefficient noted α_{trend}, to which we attributed a value of 0.9, indicating a strong preference for maintaining a consistent trajectory.
After updating tt, the agent will do the following:

- tt if low ($|tt| < 0.1$) or does not contradict the request (both the request and tt indicate the same direction), the request is applied as is.
- if tt is high ($|tt| > 0.1$) and contradicts the request (the request and tt indicate opposite directions), the agent does not apply the request, choosing instead to not change its value.
- in the special case where the request asks the agent not to change value, the agent does so whatever the value of tt.

This behavior is described in algorithm 6.12.

Algorithm 6.12: Conflicting Requests - Speed Coordination

```
$r_s \leftarrow$ selected request
// get sign of variation required by current selected request
$\Delta_r \leftarrow value_{current} - value_{r_s}$
$\sigma_r \leftarrow \begin{cases} -1 & if \quad \Delta_r < 0 \\ 0 & if \quad \Delta_r = 0 \\ 1 & if \quad \Delta_r > 0 \end{cases}$
// update trending trajectory
$tt \leftarrow$ memorized trending trajectory
$tt \leftarrow tt \times \alpha_{trend} + \sigma_r(1 - \alpha_{trend})$
// apply the coordination action (using Table 6.2)
coordination_action($tt$, $\sigma_r$)
```

Anticipating Criticality Variations of Conflicting Requests This simple comparison of the criticalities at a given instant is the main way criticality was used in previous application of the AMAS theory. Since the goal of the agents is to reduce the maximal criticality among their

Table 6.2: Conflicting Requests - Coordination Actions

tt	> -0.1	< 0.1
σ_r		
+	NOMINAL	WAIT
0	WAIT	WAIT
-	WAIT	NOMINAL

neighbors, selecting the request of the most critical neighbor is a basic heuristic to achieve this end. We provide here an additional refinement by making the agents anticipate the effects of their actions on the criticality of the other agents.
The agent will memorize the requests and observe how the criticality of the next requests vary based on its actions. From on this observation, the agent makes an estimation of the impact of its action on the criticality of the senders, using the approximation hypothesis that varying in the same direction will cause the same variation of criticality on the senders. These estimations allow the agent to make a prediction on the future criticalities of the senders in the cases where it increases or decreases.

By taking in account not only the "spacial" aspect (which neighbor has what criticality) but also on the "temporal" aspect of its action (what will be the consequence of my action on my neighbors), the agent can make a more adequate decision.

The agent deems two requests r_{max} and r, where $crit_{r_{max}} > crit_r$, to be equivalent in three cases:

- ▷ When the difference between their criticalities is lower than a given threshold t_{crit} $|crit_{r_{max}} - crit_r| < t_{crit}$.
- ▷ When the predicted difference between their criticalities is lower than t_{crit} $|(criticality_{r_{max}} + \Delta_{r_{max}}) - (criticality_r + \Delta_r)| < t_{crit}$.
- ▷ When the predicted criticality of the least critical request r is greater than the predicted criticality of the most critical one r_{max} $criticality_{r_{max}} + \Delta_{r_{max}} < criticality_r + \Delta_r$.

This behavior is presented in algorithm 6.13.

(We chose in our experiments to assign the comparison threshold t_{crit} a somewhat arbitrary value of 2% of the maximum criticality)

Each of these cases represent a situation where the two conflicting agents are reaching an equilibrium state. In these cases the agent cannot simply discriminate using only the criticality information. We will now see in section 6.3.2 an additional mechanism for such cases.

6.3.2 Cooperative Trajectories

6.3.2.1 For Variable Agents

This NCS can be seen as an extension of the previous one (conflicting requests). It occurs when a variable agent receives requests from its outputs which lead to contradictory changes on its inputs (*Conflict* NCS). We presented in the previous section how using *criticality*

Algorithm 6.13: Detecting Equivalent Contradictory Requests using Criticality Anticipation

```
$R \leftarrow$ received requests
$R_{mem} \leftarrow$ the previous memorized requests
$t_{crit} \leftarrow$ criticality comparison threshold
// calculate criticality variations
foreach $r \in R$ do
    $\Delta_c^r \leftarrow |criticality_r - criticality_{r_{mem}}|$
end
// select most critical request
$r_{max} \leftarrow \arg\max_{r \in R} criticality_r$
// find equivalent requests
$R_{eq} \leftarrow r \in R - \{r_{max}\} : \begin{cases} & |criticality_{r_{max}} - criticality_r| < t_{crit} \\ \text{or} & |(criticality_{r_{max}} + \Delta_{r_{max}}) - (criticality_r + \Delta_r)| < t_{crit} \\ \text{or} & criticality_{r_{max}} + \Delta_{r_{max}} < criticality_r + \Delta_r \end{cases}$
// update memory
$R_{mem} \leftarrow R$
```

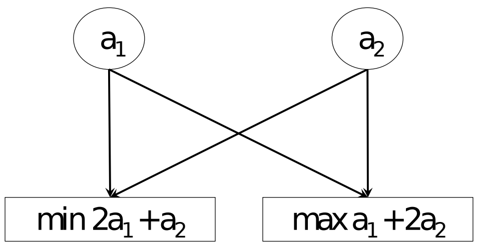


Figure 6.7: Cooperative trajectories for variable agents example.

the agent could discriminate between the requests by choosing to satisfy the most critical. However, in some specific cases, this mechanism is not the most efficient. Let us take a simple example, graphically represented in Figure 6.8.

Suppose the following problem:

$$\text{Minimize } 2a_1 + a_2$$
$$\text{Maximize } a_1 + 2a_2$$

The graphical representation of this problem is shown in Figure 6.7.

Using conflicting requests mechanisms, each variable agent would, upon receiving requests coming from the objective and constraint agents, choose to fully satisfy the most critical request, disregarding completely the other. In this example, if the request of b_1, the first objective agent, is the most critical, a_1 and a_2 would both to decrease, they would both to increase if the request of b_2, the second objective agent, was the most critical.
With this mechanism, the variable agents only have two possibilities: fully satisfy the request

of b_1, or fully satisfy the request of b_2. However we can easily observe how, in this specific problem, the input variables a_1 and a_2 have different impacts on the two objectives. A variation of 1 for the variable a_1 will change the first objective b_1 by 2 and the second one b_2 by 1, while a variation of 1 for the variable a_2 will have the inverse effect.

In such cases, the agents would start by fully helping the most critical to stabilize to a point where the criticalities of both requests are balanced., even if the point is not the most optimal.

Cooperative Trajectories using Participations This observation leads to a third possibility: trying to satisfy both requests at once by satisfying for each variable agent the request on which it has the most "impact". Doing so allows a most efficient exploration of the search space by improving several criteria at once.

This cooperative behavior intervenes only when the variable agent receives multiple contradictory requests with equivalent criticalities (*i.e.*, the agents have reached the point where the criticalities are balanced). In this case, the agent cannot discriminate between the requests using the Conflicting requests cooperative mechanism, as the requests are deemed to be equally important. When the criticalities of the contradictory agents are balanced, the helping agents must now try to improve the situations of both agents *simultaneously*. The goal is to improve the situation of both contradictory agents while keeping the equilibrium.

As the criticalities of the contradictory agents are balanced, the variable agents need an additional information in order to make an adequate action. To do so, we introduce a new measure called *participation*, propagated with the requests exchanged between agent.

The meaning of this measure is the following: when an agent A receives a request from an agent B, containing a participation p, it means that B estimated the relative impact of A on the request regarding the impact of its others inputs to be p ($p \in [0,1]$).

To estimate the participation, we base ourselves on the correlation estimation mechanism introduced in section 6.2.2. The participation of an agent given by a request is thus calculated as follow:

- ▷ Constraint/Objective agent: $participation_i = \dfrac{|correlation_i|}{\sum\limits_{j \in I} |correlation_j|}$
- ▷ Model agent: $participation_{i,o} \dfrac{|correlation_{i,o}|}{\sum\limits_{j \in I} |correlation_{j,o}|} \times participation_{request}$
- ▷ Variable agent: the *participation* of the received request is propagated as is

When a variable agent receives contradictory requests which are deemed to be of equivalent criticality, it chooses to move in the direction which for which the total participation of the requests is the highest. This behavior is described in algorithm 6.14.

This mechanism allows for several variable agents to make a coordinated action and to improve simultaneously the situation of each requesting agent. Interestingly this coordinated action is made without any explicit communication between the different variable agents, or without these agents even knowing each others existence. The participation measure provides sufficient information to the agent by giving it its relative participation, giving also an implicit information about the relative participation of the *rest of the system*.

Algorithm 6.14: Cooperative Trajectory - Variable Agent

```
$R_{equiv} \leftarrow$ selected requests of equivalent criticality
$R_{-} \leftarrow \{r \in R_{equiv} : value_r < value_{current}\}$
$R_{+} \leftarrow \{r \in R_{equiv} : value_r > value_{current}\}$
if $R_{-} \neq \emptyset \wedge R_{+} \neq \emptyset$ then
    // select direction with the highest total participation
    $Part_{-} = \sum_{r \in R_{-}} participation_r$
    $Part_{+} = \sum_{r \in R_{+}} participation_r$
    if $|Part_{-}| > |Part_{+}|$ then
        $R_{selected} \leftarrow R_{-}$
    else
        $R_{selected} \leftarrow R_{+}$
    end
end
```

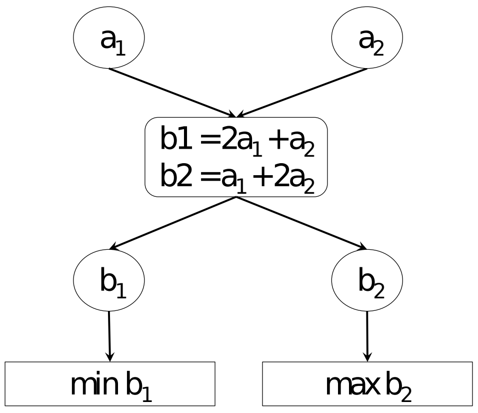


Figure 6.8: Cooperative trajectories for model agent example.

6.3.2.2 For Model Agents

We have seen in the previous section how it could be possible in some cases to propose a more efficient exploration of the search space by making variable agents cooperatively adjust their value based on the variation of the different requests they receive. However, if we take the example problem, we can reformulate it as the following equivalent problem:

$$\text{Minimize } b_1$$
$$\text{Maximize } b_2$$

Where b_1, b_2 are outputs of the model $M(a_1, a_2)$ defined as

$$\begin{aligned} b_1 =& 2a_1 + a_2 \\ b_2 =& a_1 + 2a_2 \end{aligned}$$

Using this new formulation, the model agent will aggregate the requests it receives and only transmit one request to each variable agent. Consequently the variable agents will not be able to perceive the potential cooperative trajectory. In this case, it is the model agent which must detect the possibility of a cooperative trajectory.

To this end, the model agent will use the same mechanisms than the variable agents. First it translates the requests sent by its outputs in requests on its inputs (using the classical propagation mechanisms). Then, for each input, the model agent will take the role of a variable agent and apply the cooperative trajectory mechanism for selecting a request among the requests destined to this input. After a request is selected for each input, the model agent will send the requests to its inputs agents.
This behavior is described in algorithm 6.15.

Algorithm 6.15: Cooperative Trajectory - Model Agent

```
$I \leftarrow$ the inputs set of the model agent
$R^{out} \leftarrow$ received requests
$O \leftarrow$ associated optimizer
foreach $i \in I$ do
    // translate received requests into requests on the input
    $R_i^{in} \leftarrow \{O(r^{out}), \forall r^{out} \in R^{out}\}$
    // apply variable cooperative trajectory on the input
    // (as described in algorithm 6.15)
    $r_i^* \leftarrow cooperative\_trajectory(i, R_i^{in})$
    // send the selected request to $i$
    send($i$, new Request($r_i^*$))
end
```

6.3.3 Cycle Solving

A common difficulty regarding complex optimization problem (and complex systems in general) is the existence of interdependencies (which can be thought as instances of *feedback loops*). For example, for the design of an aircraft, one could try to increase the range of the aircraft by increasing its fuel tank. But doing so, it increases the mass of the aircraft, which in return decreases its range. This complex interdependency between range and mass will appear in the analytical formulation as a cycle in the calculus of the models, *i.e.* to produce its outputs O_M, the *Mass* model will need the outputs O_R calculated by the *Range* model, taking itself in input the outputs O_M of *Mass*:

$$\begin{aligned} O_M =& Mass(O_R) \\ O_R =& Range(O_M) \end{aligned}$$

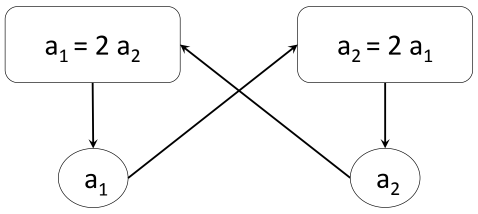


Figure 6.9: Diverging cycle example.

Such cycles in the problem formulation can pose severe challenge for the optimization process. Once more, we must keep in mind that models must be handled as black boxes; Consequently an analytical solving of the cycle is impossible. Consequently, it is the burden of the agents to detect cycle and take correcting actions of required.

The most basic situation regarding such cycles concerns two directly interdependent models (i.e each one using an output of the other as its input). However, much complex cycles topologies may be encountered. For example, an arbitrary high number of models can be successively chained together in one giant multi-step cycle, or several models can present multiple interdependencies with each other.The solution for which such a cycle is stable (if it exists) is called the *fixed point*.

As the system must work by iterating over these black boxes, the values of the outputs will be iteratively recalculated and propagated between the models involved in the cycle. Depending on the exact structure of the cycle, its effects on the problem solving can be radically different. To illustrate this, let us take the simple following example:

$$y = a1 \times x$$
$$x = a2 \times y$$

where x, y are variables and $a1, a2$ fixed parameters.

The aim is to find coherent values for x and y. Since models are black boxes, so we cannot directly infer the solution. Depending on the parameters *a1* and *a2* (and excluding the trivial case of $a1, a2 = 0$), this structure behaves differently:

- $a1, a2 = 1$: in this case any value for both x and y satisfies the problem, there is an infinity of fixed points
- $a1, a2 < 1$: in this case the system will converge toward the solution. The fixed point is said to be *attractive*
- $a1, a2 > 1$: in this case the system will diverge in the opposite direction of the fixed point, which is said to be *repulsive*

The two first cases are benign, and do not impeded the solving process. The third case however will disturb the solving process by moving the state of the system further and further from the stable solution. The task of the agent is then twofold:

1. detect existing cycles
2. if a cycle is currently diverging, take a correcting action to ensure it converges

An example of diverging cycle is shown in Figure 6.9

Detecting Cycles using Messages Signatures To address the detection of cycles, each message is uniquely signed to register its origin. When an agent initiates a message sending by itself (i.e. not as the result of a received message), it *signs* the message with its unique agent "ID" and an unique message number. The association of these two elements is the *origin signature*.

When an agent sends a message in response to a received message, it adds to the sent message the origin signature of the message causing its action. This way, the signature origin is preserved from message to message and can be used to pinpoint the origin of an activity of the system.

The output agents will be in charge of detecting and handling cycles, as they are in the best position for this. Indeed, being at the junction between criteria and models (or between different models), they are often the ones that can detect cycles the earliest.

To detect a cycle, the output agent creates a correspondence table associating for each origin agent the last signature it received from it. When receiving a message, the agent checks its origin. If the origin agent is not present in the table, the agent adds it. If it is present, it compares the new signature with the one it memorized. If the signatures are different, the agent replaces the old signature with the new one (as the new signature corresponds to a more recent request). If the two signatures match, then there is a cycle.

Finding Fixed Points In case of a cycle, the output agent must now determine if the cycle is converging or diverging toward the fixed point. As in the general case all models are black boxes, the output agent needs to observe the evolution of output values while iterating through the cycle. If the difference between successive values is decreasing, the cycle is converging toward the fixed point, else the cycle is diverging as the variables are going in the "wrong direction". To this end, the output agent memorizes the difference between its old and new value, and continues as usual. When receiving the next message from the cycle, it calculates the new difference which enables it to decide if the cycle is converging or not toward the fixed point.

If the cycle is converging, the output agent does not need to do any special action. But in case of a diverging cycle, instead of taking the requested value, the agent emits to its model a request for the opposite direction (*i.e.*, if the value of the output should have increased of n, the agent will request a new value corresponding to a decrease of n). The request will propagate through the cycle until the fixed point is found.

6.3.4 Hidden Dependencies

The fact that each agent has only a local and limited view of the problem is a requirement for the system to be able to scale to arbitrary large problems. However this restriction can prove to be problematic when, due to its limited perceptions, an agent makes a wrongful assumption regarding its neighbors. The *hidden dependency* NCS happens when an agent considers that the requests its makes to its input agents are independent while the two agents are in fact dependent. The mistake of the requesting agent comes from the fact that the dependency link between its two input agents is beyond its perception range.

Figure 6.10 is an illustration of a configuration which will lead to a hidden dependency

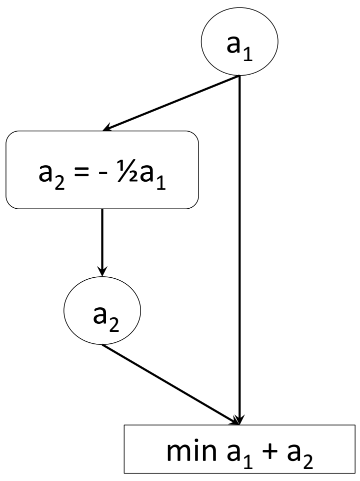


Figure 6.10: Hidden dependency example.

NCS. In this example, the objective agent min $a_1 + a_2$ will assume that the variable agents a_1 and a_2 are independent, while the value of a_2 is in fact completely dependent from the one of a_1 (more precisely $a_2 = -\frac{1}{2}a_1$). In this context, the objective agent will send separate requests to the two variable agents, asking them both to decrease. Since a_2 is an output variable, it will transmit the request to its model agent, which will in turn transmit it to a_1. And since a_2 and a_1 are negatively correlated, the transmitted request will ask a_1 to increase. Consequently a_1 will receive two requests, one (directly from the objective) asking it to decrease, and one (transmitted by a_2) asking it to increase.
This behavior is described in algorithm 6.10.

This situation is clearly a NCS, as two requests having the same origin should not require an agent to do contradictory actions. We will now propose a mechanism to allow the variable agent a_1 to solve the NCS efficiently.

Detecting Hidden Dependencies with Influences First of all the agent must identify the NCS. This part proves to be quite easy using the *origin signature* information we introduced in section 6.3.3. Since every message contains a field recording which agent was at the origin of the request, any agent can know if two requests originate from the same agent by comparing this origin field.
In our example, the agent a_1 is able to easily detect the NCS, by comparing the two contradictory requests and observing that their origins are the same.

However, our agent still does not know how to solve the NCS as the requests, originating from the same criterion, have identical criticality. To understand what is needed for the agents to be able to solve the problem, let us examine our example again.

In our example, the solution is quite obvious. We want to minimize $a_1 + a_2$, knowing that $a_2 = -\frac{1}{2}a_1$. This problem is then equivalent to minimizing $a_1 - \frac{1}{2}a_1$ or, simplified, $\frac{1}{2}a_1$. The correct behavior is thus for a_1 to decrease.
Let us now change a little this problem, considering instead a_2 to be equal to $-2a_1$. In this case the problem can be rewritten as (after simplifying) minimizing $-a_1$, or maximizing a_1. In this case the correct behavior for a_1 would be to increase.

It is now obvious that the correct behavior for the agent a_1 is extremely dependent on

the coefficient of the intermediate model. The same can be said about the coefficients of the objective, and could be said in the general case about any of the intermediate models which could happen to be between the objective and the variable agents. Conceptually, we could say that the variable agent is linked to the objective agent by different "branches", with each branch having a specific influence on the objective. In our original example the two "branches" would have influences of respectively $+1$ and $-\frac{1}{2}$ (and $+1$ and -2 in the modified example). That is, the impact of a change of a_1 of δ_{a_1} is of 1 considering the direct branch, and of $-\frac{1}{2}$ considering the branch passing by a_2, for a total influence of $\frac{1}{2}$.
If we can provide this information to the variable a_1, then the agent will be able to solve the NCS by selecting the request coming from the most influent branch(es), ensuring it action will be beneficial for the origin of the requests. We will now see how, by adding a new information with the requests, the agents are able to propagate their local influence to the variable agent, allowing it to solve the NCS.
This behavior is described in algorithm 6.16.

Algorithm 6.16: Influence propagation by internal model agents

```
behavior of Objective/Constraint Agent
    $c_i \leftarrow correlations(Inputs, \{output\})$
    // constraint and objective agents have one output only
    foreach $i \in Inputs$ do
        $influence_{r_i} \leftarrow c_i$
    end
behavior of Model Agent
    $c_{i,o} \leftarrow correlations(Inputs, Outputs)$
    // model agents can have several outputs
    foreach $r \in$ selected requests do
        foreach $i \in Inputs$ do
            $o \leftarrow sender_r$
            $influence_{r_i} \leftarrow influence_r \times c_{i,o}$
        end
    end
```

Influence Propagation Mechanism Before presenting the exact propagation mechanism, we need to make a quick clarification. In our example, we reasoned on linear models by looking at their coefficients in order to know the different influences. Such reasoning could not be applied to the general black boxes models that the agents use. However, we presented in section 6.2.2 a general algorithm which allows the agents to create local linear approximations of any black box models by observing the correlations between the inputs and the outputs of the internal model. Consequently we can assume that each agent with an internal model can always extract the linear factors corresponding to the model, either because they were given to them by the designer, or because they used the aforementioned method to create a local linear approximation (using the observed correlations). During the rest of this section, we will assume that the agents apply the estimation algorithm and use the observed correlations as approximate linear factors of the problem.

To provide the variable agent with the correct influence information, the agents use a simple propagation mechanism. An *influence* field is added to the requests transmitted into

the system. The constraint and objective agents which send the original requests fill the influence information with the observed correlation between its output and and the input corresponding to the agent to which the request is sent.
An intermediate variable agent will propagate the request without modifying this field, while a model agent will replace it by its old value *multiplied* by the correlation its observed between the output from which it received the request and the input to which the request is sent (this operation is done independently for each input if a request is propagate to several inputs of the model).
When a variable agent detect a hidden dependency NCS, it separates the requests coming from the origin in two groups: the requests asking the variable to increase, and the requests asking the variable to decrease. For each group the agent calculates the sum of the influences of each request in the group, then compare the *absolute values* of the total influences. The agent will select the action corresponding to the biggest influence (in absolute value). If the agent is a design variable, it will change accordingly, if it is an output variable it will propagate a request with the same origin and information, but replacing the influence value with the sum of the influences of the winning group.
We can say that, in the same way that we “collapsed” the intermediate models when we wanted to find the solution to our example problem, the intermediate correlations are “collapsed” into the influence information for the variable agent to be able to solve the NCS.

The detection and solving of a hidden dependency NCS by the variable agents is described in algorithm 6.17.

Algorithm 6.17: Hidden dependency detection and solving by variable agents

```
$R_{received} \leftarrow$ received requests
// select most critical request
$R_{selected} \leftarrow \arg\max_{r \in R_{received}} criticality_r$
if $|R_{selected}| \geq 2 \wedge \forall(r_1, r_2 \in R_{selected} : origin_{r_1} = origin_{r_2})$ then
    $R_- \leftarrow \{r \in R_{selected} : value_r < value_{current}\}$
    $R_+ \leftarrow \{r \in R_{selected} : value_r > value_{current}\}$
    if $R_- \neq \emptyset \wedge R_+ \neq \emptyset$ then
        // a conflicting hidden dependency NCS is detected
        $Inf_- = \sum_{r \in R_-} influence_r$
        $Inf_+ = \sum_{r \in R_+} influence_r$
        if $|Inf_-| > |Inf_+|$ then
            $R_{selected} \leftarrow R_-$
        else
            $R_{selected} \leftarrow R_+$
        end
        handle $R_{selected}$
    end
end
```

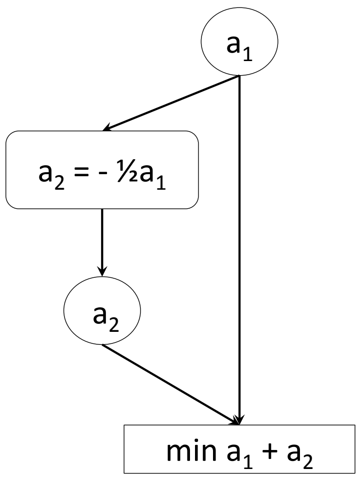


Figure 6.11: Hidden dependency example (reproduced).

6.3.5 Asynchronous Messages

The last NCS addresses an underlying assumption we have made until now. When we described the behavior of the agents, we supposed each of them always receive the messages in a timely manner, waiting if necessary until all the relevant messages are transmitted before acting. Once again, this hypothetical scenario is made impossible by the need for the agents to maintain a local view. The agents do not know what messages are currently transiting in the system. Their only knowledge concerns the messages they received at the start of their cycle and what messages they previously sent (but not their current status). Consequently, the topology of the agent graph can lead to transmission delays, where messages sent at the same time by the same agent will not necessarily reach their ultimate destination at the same time. For example, let us look back at the hidden dependency example (reproduced on Figure 6.10 for convenience). When we discussed the case of hidden dependencies, we assumed by all the messages sent by the objective agent would reach a_1 before the latter would take its decision. However, it is clear that one of the message will take more time to reach a_1 than the other, as it must transit by two intermediates. Why would the agent a_1 wait for this late message, without even knowing its existence, before acting?

Many of the previous mechanisms we proposed rely on a timely information delivery, for example for the agents to be able to detect some NCSs. Based on this, it results that, without an adequate "synchronization" mechanism, many of them would fail to work correctly. On the other hand, having a fully synchronized system, where all the agents would make lengthy checks to ensure the complete propagation of all messages to all recipients, would be undesirable. Indeed, complex continuous optimization problems create big but loosely connected graphs, for which such full synchronized mechanism would be both costly and inefficient. Moreover, if we take in account the dynamic aspects of the problem (having a designer which can add or remove agent at any moment during solving), obtaining such synchronization guarantee can be nearly impossible.

Asynchronous Messages with Influences and *allgood* Messages We propose a somewhat middle ground mechanism, where the agents try to estimate if they received enough messages to make a correct decision. As some of the information they require is learned during the solving process, an adaptation period will be necessary in order for the agent to behave correctly. During this adaptation period the agents may have a suboptimal behavior where some NCSs are not correctly detected. The iterative nature of the solving process allows these suboptimal decisions to be ultimately corrected by the agents themselves.

The solving mechanism for this algorithm is composed of two parts, regarding how the agents make their estimations for respectively the request messages or the inform messages they expect to receive.

1. Concerning the requests they expect to receive, the agents base their reasoning on the fact that requests are only sent in answer to previously received inform messages. Consequently they except an answer for the inform messages they previously send. So the agent assume at least one request message by output agent to which it sends an inform message. As the agents are cooperative, each output agent should forward at most one request (the one deemed the most important to handle) back to the agent. Consequently, an agent can expect to receive one and only one request message in response to the informs it send. Once it received a request message from each of its output agents, it can take its decision (either forwarding the most important request to its input(s) or, in the case of a design variable, changing its value and sending new inform messages).
   The only modification which needs to be done for this mechanism to work concerns (non-criterion) agents which are not linked to any output agents. Previously, when receiving inform messages, these agents would change their value accordingly, but would not send back any message (as they have no request to do). In order for the "upstream" agents not to be blocked waiting for an answer, an agent that neither has requests to send or informs to forward must answer to its input agent with an *allgood* message, signaling that it has taken into account the inform message and that non other action is pending. An agent receiving such *allgood* message knows that it no longer needs to wait for this output for taking its decision.
   This behavior is described in algorithm 6.18.

2. In the same way that an agent waits for requests messages after having sent informs, an agent which has send request messages should wait to receive enough relevant inform messages before taking further steps. In the same way that an agent sending informs expects to receive request messages in return, an agent sending requests expects to receive inform messages in response. A basic solution is thus to apply the same algorithm. In this case however, the agent is not strictly required to wait for all the inform messages to take a relevant decision. We presented in section 6.2.2 how the internal model agents can get an indication of the impact of their inputs in regard of their outputs using correlation information. Consequently, such agents can wait until they estimate that they received informs from their inputs corresponding to a total of more than 50% of the total "impact" on the value of each output (variables agents, having only one input, always have to wait for this input and this input only, which correspond of 100% of the total "impact" on its value).
   This behavior is described in algorithm 6.19.

In our implementation, we did not concern ourself with potential loss of messages, which can happen when the designer intervenes on the system by removing existing agents while

Algorithm 6.18: Waiting algorithm for request messages

```
$O \leftarrow \{$ outputs $\}$
$Requests \leftarrow \{$ received requests $\}$
$AllGoods \leftarrow \{$ received allgoods $\}$
$stillwaiting \leftarrow |Requests| + |AllGoods| \neq |O|$
```

Algorithm 6.19: Waiting algorithm for inform messages

```
$O \leftarrow \{$ outputs $\}$
$Informs \leftarrow \{$ received informs $\}$
$stillwaiting \leftarrow \forall output \in O, \sum_{info \in Informs} influence(input_{info}, output) \leq 50\%$
```

messages are transiting. For a "production-ready" version of the algorithm, such potential losses should be taken into account. Otherwise some agents may hang indefinitely waiting for lost messages. A simple way to remediate to this problem is to add timeouts to the agents. Timeouts are a well-studied problem in computer networks, and adequate algorithms can be proposed based on the works in the existing literature (see [Jac88] for example).

6.3.6 Summary of Non-Cooperative Situations

In this section we presented several problematic configurations (NCSs) which can arise from the specificities of our modeling of a continuous optimization problem. For each difficulty we presented a basic example of problematic configuration and how this configuration would cause a cooperation failure with the nominal behavior of our agents. The different NCSs are summarized in Table 6.3

In each case, we proposed general mechanisms for the agents to be able to detect and correct the NCSs in order to maintain a correct and efficient optimization process. In general, we had to provide enough information for the involved agents to detect and correct the NCS. The different mechanisms are summarized in Table 6.4

Table 6.3: Non Cooperative Situations Summary

NCS	Description	Cooperative Mechanisms
Conflicting Requests	An agent receives several incompatible requests	Criticality
Cooperative Trajectories	An agent receives seemingly incompatible requests, which can each be satisfied	Participation
Cycle Solving	Several agents are involved in a diverging cycle	Signature
Hidden Dependencies	An agent sends seemingly independent requests to dependent agents	Signature, Influence
Asynchronous Messages	Agents receive messages in untimely manner	Influence

Table 6.4: Non Cooperative Situations Solving Mechanisms

Mechanism	Description	Properties
Criticality	A aggregated measure to indicate the state of the agent	Comparable between different agents
Signature	A unique signature composed of the unique id of the sender/origin of the message and a (local) timestamp	Comparable, allows a partial ordering of the messages by sender/origin
Influence	An indicator of the impact value of the receiver on the origin	Comparable by origin
Participation	An indicator of the relative impact of the receiver on the origin relative to the rest of the system	Comparable between different senders

Complexity Analysis of Solving Mechanisms

We now provide a quick complexity analysis of the different solving mechanisms.

Conflicting Requests The main work for solving this NCS consists in finding the request(s) with the highest criticality. This very simple operation can be done by examining each received request in sequence and have a complexity of $\mathcal{O}(n)$ in the number of requests.
The solving of this NCS does not require sending additional messages

Cooperative Trajectories When solving this NCS, the agent must group the requests before comparing the total participation of the two groups. The grouping and total participations calculation can be factorized into a single pass algorithm of complexity $\mathcal{O}(n)$ regarding the number of requests. The agent will not need to send additional messages for solving this NCS.

For models agents, the algorithm will need to be applied on each input, consequently its complexity will be $\mathcal{O}(n \times i)$, i being the number of inputs.

Cycle Solving To detect a cycle, the agent needs to compare the signature of each received message with the memorized origin signatures. The complexity of this comparison is $\mathcal{O}(n \times m)$, n being the number of received messages and m being the number of signatures.

Hidden Dependencies When solving this NCS, the agent needs to detect which selected requests come from the same origin, group them, before comparing the total influences of the two groups. The detection, grouping and total influence calculation can all be factorized into a single pass algorithm of complexity $\mathcal{O}(n)$ regarding the number of requests.
The agent will not need to send additional messages for solving this NCS.

Asynchronous Messages The algorithm for waiting request is trivial and can be executed in constant time. The algorithm for waiting informs requires a study of every inform for each output, its complexity is thus $\mathcal{O}(n \times o)$, n being the number of informs and o the number of outputs.

This mechanism requires the sending of additional messages from (non-criterion) output agents which themselves do not have any output agent. In the best case, there is no such agent, thus no additional message is sent. In the worst case all outputs agent are concerned, and need each to send one additional message (message complexity of $\mathcal{O}(o)$, o being the number of outputs).

It can be seen that, overall, the cooperative mechanisms we proposed to solve the various NCSs are quite reasonably efficient in their complexity. This property is important in order for the system to be able to scale to bigger and more complex optimization problem. We managed to keep the complexity of the algorithms low by keeping the reasoning of the agents to a local level, limiting their perceptions to their neighborhood and only very synthetic information about the rest of the system (criticality, origin *etc.*). Moreover, it should be noted that, in many problems, some of the agents will only have a minimal role in the solving, being connected only to one agent in input and one agent in output. For these agents, the complexity of these mechanism will be minimal as they will always receive at most one message. Only in the utmost complex and coupled problems will the majority of the agent be involved in the solving of NCSs.

A concern could be raised about the composition of these different mechanisms. Indeed an agent can be involved in several NCSs at once. In this case, would not the combination of these mechanisms cause exponential increase of complexity? While this concern could be correct in theory, in practice the different solving mechanisms play the role of successive filters regarding the received messages. Consequently, later mechanisms work on a far lower number of messages than the earlier ones. The typical example is the conflicting requests mechanisms, which will eliminate all the requests which are not deemed critical enough. The others mechanisms will only have to take in account the requests which were conserved by this first filter. More so, it should be noted that, while the different mechanisms have be presented separately, a large part of the processing they do on the messages can be factorized, in order to increase the overall efficiency. While in our implementation we did not do such optimization (in order to keep the mechanisms separated and code easier to work with), such improvements would be of course strongly recommended for a more "production-ready" implementation of the system.
To conclude on this remark, we can safely say that the composition of the different cooperative mechanisms is not of additive complexity, but only bring a marginal increase in the total complexity of the agent behavior.

7 Extending the Multi-Agent System for Uncertainties

We have seen in section 3.3 how uncertainties can be a major concern in real-life optimization. We have also seen how several types of uncertainties existed and needed to be taken into account. Moreover, complex optimization problems can involve several teams concerned with different disciplines, each of which with its own practices regarding how to manipulate uncertainties, making difficult to study the impact of uncertainties across disciplines.

In this chapter we present a way to alleviate this burden by having the agents automating the uncertainty propagation in the system.

The difficulty is twofold:

- ▷ **How do the agents handle uncertainties ?** How does a model agent calculate the uncertainties on its outputs from the uncertainties of its inputs and its internal model ? How does a constraint agent can handle a probabilistic constraint ?
- ▷ **How do the agents combine heterogeneous uncertainties modeling ?** How can a model agent handle inputs with different uncertainties modelings ?

To answer these difficulties we will present a general mechanism inspired by the one we used with external optimizers, which is based on the use of encapsulated external tools to apply the domain-specific expertise.

7.1 From Deterministic Optimization to Optimization under Uncertainties

We previously explained that uncertainties can be present at several levels in the optimization problem. We consider here the two following types of uncertainties:

- ▷ **Variable uncertainties**, which are brought by the fact that we cannot perfectly control the value which will be effectively assigned by the design variable.
- ▷ **Model uncertainties**, which represent the limited precision of the models.

Taking in account these uncertainties leads to a modification of the criteria formulation. Indeed, the constraint $x < 10$ makes sense in the context of deterministic optimization, but not so much when the value of x is uncertain. Is the constraint satisfied when x is an uncertain value, for example represented by an interval ranging from 8 to 12?
One can instead propose the following probabilistic constraint $P(x < 10) > 90\%$, which can

still be evaluated when manipulating uncertain values. It is possible to propose others criteria formulations manipulating uncertain values, for example one could consider a deterministic constraint concerning the mean of an uncertain variable. Usually, constraints are expressed in a probabilistic form, while objectives use the mean of the variables.

7.2 Manipulating Uncertain Values

Taking in account uncertainties requires modifications at several levels. The most obvious concerns the data structure. On the deterministic version, the data exchanged by the agent were simply real values [1]. To be able to exchange uncertain values, the agents require a more complex data structure.

As we presented earlier, there are many ways to represent uncertainties, each of them having specific requirements. For example, an interval representation requires two values, the minimum and maximum boundaries, while an uncertainty expressed by a set of measures obtained with Monte Carlos experiments can require the storage of hundreds of thousands points. As we want to maintain the possibility for the designers to choose how they represent uncertainties, we need to provide an abstraction for the agents to be able to consistently handle uncertain values whatever the specific representation.

To this end, we specified the minimum set of operations the agents require. To be able to manipulate an uncertain value, an agent needs to be able to :

▷ obtain the mean of the value (or at least a representative value).
▷ obtain the standard deviation of the uncertainty.
▷ draw a random value following the law represented by the uncertainty.
▷ obtain the cumulative probability of a value (*i.e.* for a given value t, what is $P(X < t)$)
▷ obtain the inverse cumulative probability (for a given probability p, what is t for which $P(X < t) = p$) associated with the uncertainty).

Any uncertainty representation which can provide these five informations can be used by the system. All the designer needs to do is to provide the module which implements these operations[2].

Let us see how one can provides these information regarding two drastically different modelings: intervals and Monte Carlo experiments.

Intervals Representation As the reader knows, an interval representation consist of a lower and upper boundaries. It can be considered as an uniform law, with all the values between the boundaries considered as equally likely. Suppose l and u the boundaries of our uncertainty, we can obtain the required informations using well-known formulas for the uniform law.

▷ the mean can easily be obtained as $\frac{u+l}{2}$.
▷ the standard deviation is $\sqrt{(u-l)^2/12}$.
▷ the drawing of a random value consists of drawing a value between l and u using an uniform law, which presents no difficulty.

[1]With the obvious limitation of the representation of real numbers on a machine.

[2]The reader familiar the concepts of object-oriented programming and interface implementation will have no difficulty to guess how the process is done in our prototype

- ▷ the cumulative probability of t is calculated as $P(X < t) = \begin{cases} 0 & \text{if t < l} \\ \dfrac{t-l}{u-l} & \text{if } l < t < u \\ 1 & \text{if t > u} \end{cases}$
- ▷ the inverse cumulative probability of p is obtained as $t = l + p(u - l)$.

As we can see, for a simple representation as intervals the required informations can be obtained very easily.

Monte Carlo Experiments The Monte Carlo Experiment is not an uncertainty law *per se* but a series of random drawings to obtain experiment points. If the number of drawings is big enough, then the series can provide an adequate information about the uncertainties governing the variable. Suppose $\{x_n\}$ a series of n drawings over x. We suppose the values to be ordered (that is, $x_i < x_j$).

- ▷ the mean can be obtained with $\frac{1}{n}\sum_n x_n$.
- ▷ the standard deviation is obtained by first calculating the variance using the formula $Var(X) = E(X^2) - E(X)^2$ (possible as we can calculate the mean), then taking the square root to obtain the standard deviation.
- ▷ the drawing of a random value consists of drawing a value i between 1 and n and returning the corresponding point x_i
- ▷ the cumulative probability of t is calculated as $P(X < t) = \dfrac{|\{x_i, \forall x_i < t\}|}{|\{x_n\}|}$ (where $|\{x\}|$ is the cardinality of the set $\{x\}$), *i.e.* the number of elements lower than t divided by the total number of elements.
- ▷ the inverse cumulative probability of p is obtained as $t = x_i$, where $i = p \times n$.

Even with very specific, non-analytical representation of uncertainties such as datasets from Monte Carlo experiments, we can provide the needed informations without much hassle.

Now that we have defined a common set of operations common to all the uncertainties representations, we can adapt the agents behavior in order to handle them. Such "uncertainty container" can be associated to variables to represent uncertain values, or to models, indicating uncertainties concerning the underlying model.

For example the designer can estimate that, due to physical limitations, the value of a design variable x cannot be exactly be controlled and will always suffer from an imprecision of +/-0.1 around its assigned value. He can model the uncertainty around the variable with an interval. If he decides to arbitrarily initialize the variable at the value 8, the uncertain value of x will then be [7.9; 8.1].

If the designer observe that the real-world process represented by a specific model tends to produce outputs whose values imprecision can be accurately modeled by a normal distribution, he can associate the normal distribution uncertainty to the model. For example the model $y = 2x$, upon the reception of the information $x = 1$, will produce an uncertain value y with a mean of 2 and a variance corresponding to the associated normal law.

We have now seen how the agents can manipulate uncertainties in isolation. But these mechanisms do not allow the agents to propagate uncertainties in the system. Indeed, if

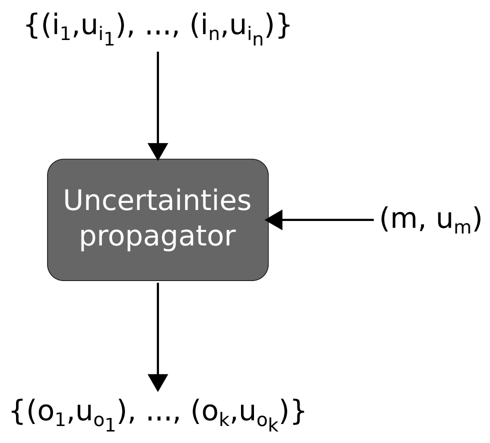

$\{(i_1,u_{i_1}), \ldots, (i_n,u_{i_n})\}$
Uncertainties propagator
(m, u_m)
$\{(o_1,u_{o_1}), \ldots, (o_k,u_{o_k})\}$

Figure 7.1: Uncertainties propagator as a black box function.

we take our two previous examples, what if the variable x ,associated with an interval uncertainty, is assigned in input of the model $y = 2x$, following a normal distribution? The model has no mechanism to handle uncertainties *received on its inputs*, but these uncertainties must be taken into account as they surely have consequences on the uncertainties of its outputs.
The difficulty here is twofold. Not only the model agent must be able to take in account the uncertainties provided by its inputs in addition to its own uncertainty, but it must be able to do so even with uncertainties of different kinds (in our example, interval and normal distribution). We will now present a generic mechanism using *uncertainties propagators* which allows to solve both problems at once.

7.3 Uncertainties Propagators

In the same way we encapsulated expert knowledge on numerical optimization using external optimizers, we propose to encapsulate the knowledge related to the propagation of uncertainties into dedicated modules we call *uncertainties propagators*. While external optimizers are used by the agents to solve local optimization problems, the uncertainties propagators are used by the agents to propagate the uncertainties associated with their inputs and model into uncertainties on their outputs.

The role of an uncertainties propagator is to determine the uncertainties associated with the outputs values of the model based on:

▷ the model itself
▷ the uncertainty associated with the model
▷ the input values
▷ the uncertainties associated with the inputs

As the outputs values cannot be determined by simply evaluating the (deterministic) model, the agent in charge of it will delegate the whole evaluation process to the uncertainties propagator when working with uncertainties. From the agent point of view, an uncertainties propagator is a black box function taking uncertain input values and an uncertain model, and producing uncertain outputs values.

Figure 7.1 illustrates the representation of an uncertainties propagator from the point of view of an agent. The uncertainty propagator is a black box. The (i_j, u_{i_j}) represent the n uncertain input values of the model (the i_j corresponding to the mean value of the input and u_{i_j} to the associated uncertainty). The model itself m is also provided along with its associated uncertainty u_m. Based on the inputs and the model (and their uncertainties), the uncertainty propagator returns the (o_j, u_{o_j}) represent the k uncertain output values produced.

This approach shows several advantages. First it allow to keep the functioning of the agents nearly untouched, which in itself is a good sign. Handling uncertain values can be deemed to be an orthogonal concern to the previous ones, and orthogonal concerns should lead to orthogonal changes. Thus this modification of our system to handle uncertainties show that *our modeling is flexible and can be extended easily*.
This approach also permit to reduce the work of the designers, which only need to *define the needed propagators once and reuse them* on several problems. This aspect is particularly important as defining a correct way to propagate uncertainties across different disciplines can be an extensive work involving experts from different domains. Providing a way to encapsulate the result of this work in a reusable component leads to a more efficient work process (this concern is especially recognized by software engineers, which created the well-known acronym: DRY - “Don’t Repeat Yourself”).
Finally, by strictly encapsulating the required knowledge, we provide a highly modular and adaptable mechanism. *If the needs regarding the modeling of the uncertainties changes, the consequences on the problem are limited solely to the uncertainties representation and/or propagators*. For example, if a designer needs to change the uncertainties associated with a model, he only has to change the uncertainty representation of the model and to use the appropriate propagator (or create it if no adequate propagator already exists).

7.4 Conclusion on Uncertainties Handling

We have seen how the agents can automatically propagate uncertainties in the system, relieving the designers of a part of the burden.
This extension to our modeling allow to take in account not only values but uncertain values, that is, values with an associated uncertainty representation. As the exact representation does not concern the agent, these uncertainties are mostly handled as black boxes. In a similar way to how they use optimizers to propagate requests with black box models, internal model agents can use uncertainties propagators to locally propagate uncertainties. In this regard, uncertainties propagators (as well as external optimizers) can be seen as a way to extend the capabilities of the agents provided by the users.

While this mechanism require for the users to provide the uncertainties propagators, it has the advantage that this specific effort has to be done only once, as the resulting propagator can be reusable at will. In this way it can greatly reduce the time different teams need to spend together to overcome the difficulties induced by different working practices. It also open the possibility of propagators libraries, which can be shared and imported as needed.

A note of warning however, the representation of uncertainties is inherently imprecise. In the same way that the modeling of a system contains imprecision, so does the modeling of the uncertainties used to represent this very same imprecision (and so would do the uncertainties used to represent the imprecision on the uncertainties...). Like discussed before,

this imprecision is an inherent limitation of the real world. As the uncertainties modeling is in itself factor of uncertainties, it follows that the transformation between two different uncertainties representations will in itself contains some part of imprecision (if only because the passage from one representation to another can lead to information loss, *e.g.* passing from a normal law to an interval modeling).
This problematic is not specific to our technique and is indeed present all the same if the uncertainties propagation is done “manually”. But it is worth reminding, as this complete automation of the propagation could hide it from the mind of the designer.

As a remark we have seen how, by overloading some common mathematical operators, it was possible for the agents to manipulate uncertain values as if they were simple numerical values. Such work could be possibly done with complex numbers, matrix or others, arbitrarily complex, data structures, as long as the corresponding overloaded operators are provided.

Synthesis of the Contribution

In this part we presented a novel approach for complex continuous optimization using a multi-agent system do distribute the optimization process. We started our work with a new way to model a continuous optimization problem as an agent graph. This modeling allows for an easy and potentially automatic transformation of any problem as a graph, without requiring special knowledge or expertise regarding MAS. Moreover, this transformation does not requires any specific reformulation of the problem, which can keep the natural formulation corresponding to its domain. Because of these properties we named our modeling Natural Domain Modeling for Optimization (NDMO).
To the best of our knowledge, this is the first proposal for providing a common modeling of continuous problems as agents graphs. While we cannot guarantee that is specific modeling will be adopted, it is surely a first step toward opening the field to other contributions. One could wonder if, by dividing the problem into different entities, NDMO would not fall into the same pitfall than MDO methods, applying to the problem a reductionist approach. However, a fundamental distinction is that MDO methods *modify the initial problem in order to reduce its complexity*, while our modeling does not make such modification. On the contrary this modeling fully conserve the interdependencies between the different elements, maintaining an integral and integrative formulation of the problem.

Based on this modeling, we proposed a simple nominal behavior where the agents try to solve individual goals and only has a limited, local perception of their neighbors. These agents can interface with external optimization techniques to solve local optimization problems (or alternatively use some internal optimization mechanisms), and exchange request and inform messages in order to keep a global coherence of the solution. A major distinction with MDO methods is that the global coherence of the problem is not a separate phase but is fully taken in account during the entire process.
While our agents are already able to interface with external optimization methods, some refinements could be proposed. An example could be to, instead of providing each agent with an adequate optimizer, providing all the agents with an optimizer catalog where several optimization methods are presented and characterized. Based on their observation of the manipulated models, the agents could select the optimization method they deem the most appropriate, and even switch between different methods during the solving. Such mechanism would contribute even more to relieving the burden of the designer, but would require to be effective an extensive study of the ontological properties of the field. An other improvement could concerns the internal optimization mechanisms we presented in algorithm 6.7. It could be possible to replace our simple linear approximation estimation by more astute techniques. Such methods could provide more precise estimations to be used by the agents.

This nominal behavior being insufficient to solve some of the issues brought by the specificities of continuous optimization, we use the AMAS theory to identify a set of non cooperative situations, which represent specific patterns of configurations where the nominal behavior of the agents result in a sub-optimal handling of the optimization process. For each of these NCSs, we proposed cooperative mechanisms which enrich the nominal behavior of the agent. These additional mechanisms allow to, when necessary, detect and identify the NCSs, and to do corrective actions in order to re-establish the correct optimization process.
The AMAS theory has proved to be an effective guide to the conception of our system. By maintaining a local view and concentrating on specific problematic situations, we obtained a behavior which is both scalable and modular. An interesting aspect of this work concerns the different NCSs we identified. As our application domain is in itself quite abstracted and general, the NCSs we encountered seem themselves to be of a more general nature than in some other works. In chapter 9 we will explore possible uses of this work to other domains than continuous optimization.

We then showed how both our modeling and solving mechanisms are modular enough to be extended in order to handle additional concerns by proposing some modifications which allow to take in account the handling of uncertainties during the optimization process. We detailed the requirements for the agents to be able to handle uncertain values instead of normal numerical values, and we introduced the new concept of uncertainty propagator to provide, using the designers expertise of the uncertainties, an automatic mechanism for propagating heterogeneous uncertainties in the system.
Uncertainties handling is an important concern in design optimization. As design optimization problems are among the most complex continuous optimization problems, it seemed important to us to provide such functionalities for the system. A quite interesting work would be to propose new ways to extend the system, for example replacing numerical values by vectors, or by extending model agents to manipulate multiple models of different fidelity, experimenting with the possibilities (and the limits) of our proposed modeling.

Design, Implementation and Extending the AMAS4Opt Building Blocks

The work presented in this thesis is part of the ongoing ID4CS project. ID4CS (*Integrated Design for Complex Systems*) is funded by the French *Agence Nationale de la Recherche* (National Research Agency)[3] and involves nine partners, both from the academic and industrial world. The goal of the ID4CS project is to use the MAS approach to provide new tools for the design of complex systems. Our industrial partners, Airbus and Snecma, are involved in aeronautical design, and are thus fully concerned by the problematic of complex continuous optimization for system design.

For this reason, one of our goals was not only to propose a new approach for continuous optimization, but also to put this approach in practice by integrating it into a working prototype which could be used by our partners. To this end we worked with our partners, including optimization specialists and software development experts, to propose such a tool, basing ourselves upon the ADELFE method. This method aims to guide the development of AMAS-based softwares from high level user requirements to the implementation nuts and bolts. Using ADELFE we made a comprehensive analysis of the domain and actors involved in the use of such a tool.

We also use the design tools provided with ADELFE to instantiate our MAS and integrate it into the prototype, among which the recent MAY framework. The goal of MAY is to provide suitable abstractions as well as reusable software components for the development of agents and multi-agents systems. We contributed to the enrichment of its components library by developing a general and modular agent architecture consistent with the AMAS theory, notably by proposing a *modular skill stack principle*, where different skills can be composed to address specific requirements.

While such work could seem to concern software engineering experts more than artificial intelligence specialists, we will see how existing MAS oriented methods such as ADELFE are still too high-level to be successfully directly applied to any domain (and more so to continuous optimization) without an important agent expertise and extensive research work. We will detail how the scientific work we produced and presented in the previous part, especially the identification of various NCSs and the mechanisms to solve them, can be generalized into "building blocks". These *building blocks* could then be reused to guide the design of other MAS in the domain of problem solving. This work places itself into a more general effort to provide a general reusable toolbox for assisting non experts in applying AMAS to the domain of optimization, under the name *AMAS4Opt* (AMAS for Optimization).

[3]COSINUS program, ANR-09- COSI-005 reference

8 ADELFE and MAY Architecture

8.1 Overview of ADELFE

In section 4.2.4, we succinctly presented ADELFE[1], the method which has been proposed for the design of AMAS. In this chapter we get back in more details about this method and what benefits it provided for the design of our system.

ADELFE is a method devoted to software engineering of adaptive multi-agent systems. It was initiated by a French government funded project lead by IRIT (see `http://irit.fr/ADELFE-Project` for details) and has been continuously enhanced since. Like previously said, the name "ADELFE" is the French acronym for "toolkit to develop software with emergent functionality" (*Atelier pour le DEveloppement de Logiciels à Fonctionnalité Emergente*).

The ADELFE method in itself is based on the Rational Unifed Process (RUP) and is defined following the Software Process Engineering Meta-Model (SPEM) [PG04; Ber+05]. Since its revision [Rou08], it is composed of five *Work Definitions* (*WD*), themselves decomposed in several *activities* making use of UML as well as the AMAS-ML and muADL languages. An overview of the method is shown on Figure 8.1.

In regard of the RUP, ADELFE adds supplementary activities and roles which are specific to its approach for the design of AMAS compliant software.
The final requirement study (WD_2), was complemented with activities 6 and 7-2 concerning the characterization of the system environment and the identification of cooperation failures during the determining of the use cases. During the analysis (WD_3), additional activities 11 and 12 respectively check for the adequacy of AMAS to the problem and identify the agents involved in the system being built, while activity 13 is complemented with a step concerning the study of the relationships between agents. During the design (WD_4), activities 15 et 16 concerns the design of the system and the agents, while activity 17 concerning fast prototyping was added in order to be able to quickly test the proposed behavior of the agents. The development phase ($WD5$) concerns the architecture and the implementation of the agents and the system.

[1]`http:/www.irit.fr/ADELFE/`

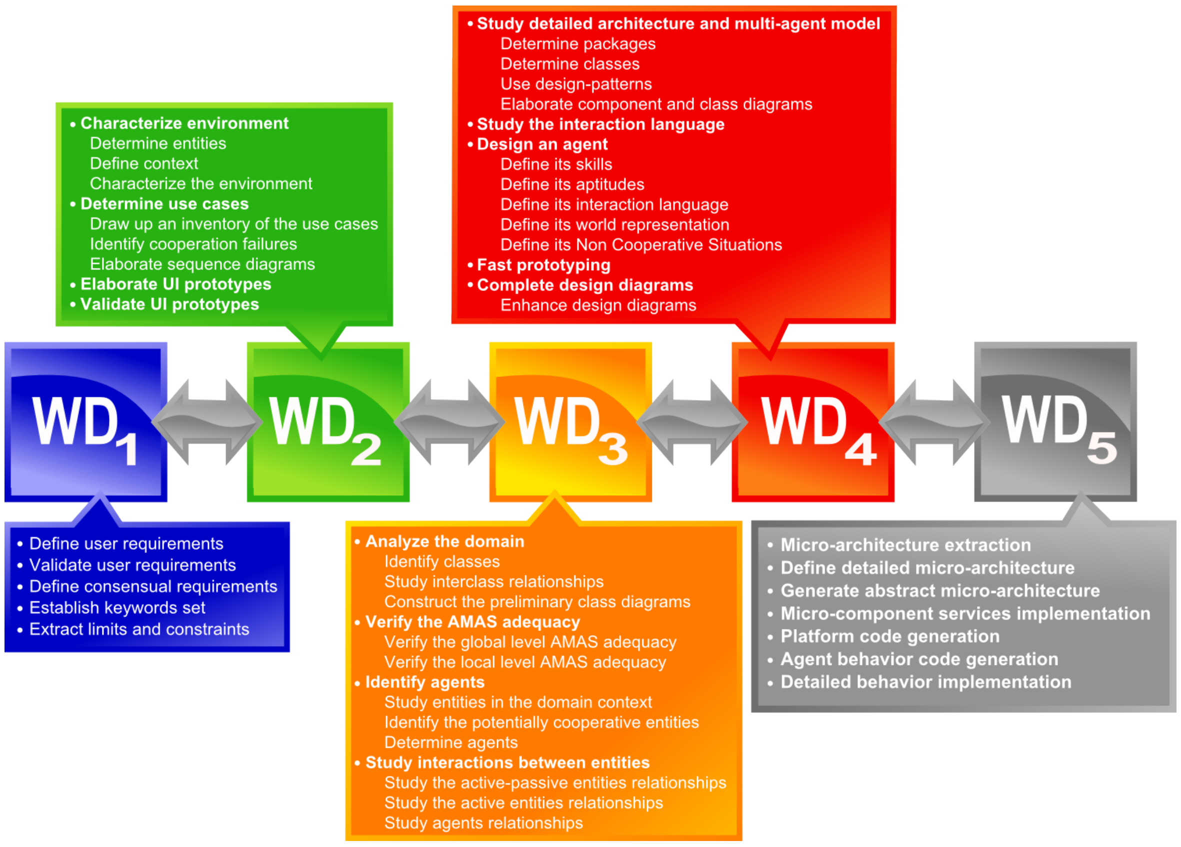

• Characterize environment
Determine entities
Define context
Characterize the environment
• Determine use cases
Draw up an inventory of the use cases
Identify cooperation failures
Elaborate sequence diagrams
• Elaborate UI prototypes
• Validate UI prototypes
• Study detailed architecture and multi-agent model
Determine packages
Determine classes
Use design-patterns
Elaborate component and class diagrams
• Study the interaction language
• Design an agent
Define its skills
Define its aptitudes
Define its interaction language
Define its world representation
Define its Non Cooperative Situations
• Fast prototyping
• Complete design diagrams
Enhance design diagrams
WD1
WD2
WD3
WD4
WD5
• Define user requirements
• Validate user requirements
• Define consensual requirements
• Establish keywords set
• Extract limits and constraints
• Analyze the domain
Identify classes
Study interclass relationships
Construct the preliminary class diagrams
• Verify the AMAS adequacy
Verify the global level AMAS adequacy
Verify the local level AMAS adequacy
• Identify agents
Study entities in the domain context
Identify the potentially cooperative entities
Determine agents
• Study interactions between entities
Study the active-passive entities relationships
Study the active entities relationships
Study agents relationships
• Micro-architecture extraction
• Define detailed micro-architecture
• Generate abstract micro-architecture
• Micro-component services implementation
• Platform code generation
• Agent behavior code generation
• Detailed behavior implementation

Figure 8.1: Overview of the ADELFE Method.

8.2 Applying ADELFE for the Design of a Continuous Optimization Tool

Based on the guidelines of the ADELFE methodology we defined the requirements of the tool we proposed to develop. The following functionalities were identified:

- ▹ Our tool will allow users to solve multidisciplinary optimization problems.
- ▹ The user will be able to load disciplinary models into the tool, express constraints and objectives on these models and then use and interact with the tool to solve the defined problem.
- ▹ Our tool will be reusable from one optimization problem to another in different application domains and will be a generic optimization tool.
- ▹ The user will be able to express uncertainties on the models or variables.
- ▹ The solving will take into account these uncertainties.
- ▹ The tool will be able to integrate well-known optimization techniques to be used locally on the models constituting the global problem. It will allow the user to add new techniques to use during solving. The tool will also be able to interact with others optimization tools to ease the optimization process.
- ▹ The user will be able (at runtime):
  - to monitor the system activity,
  - to observe time history qualitative indicators,
  - to adjust its parameters.

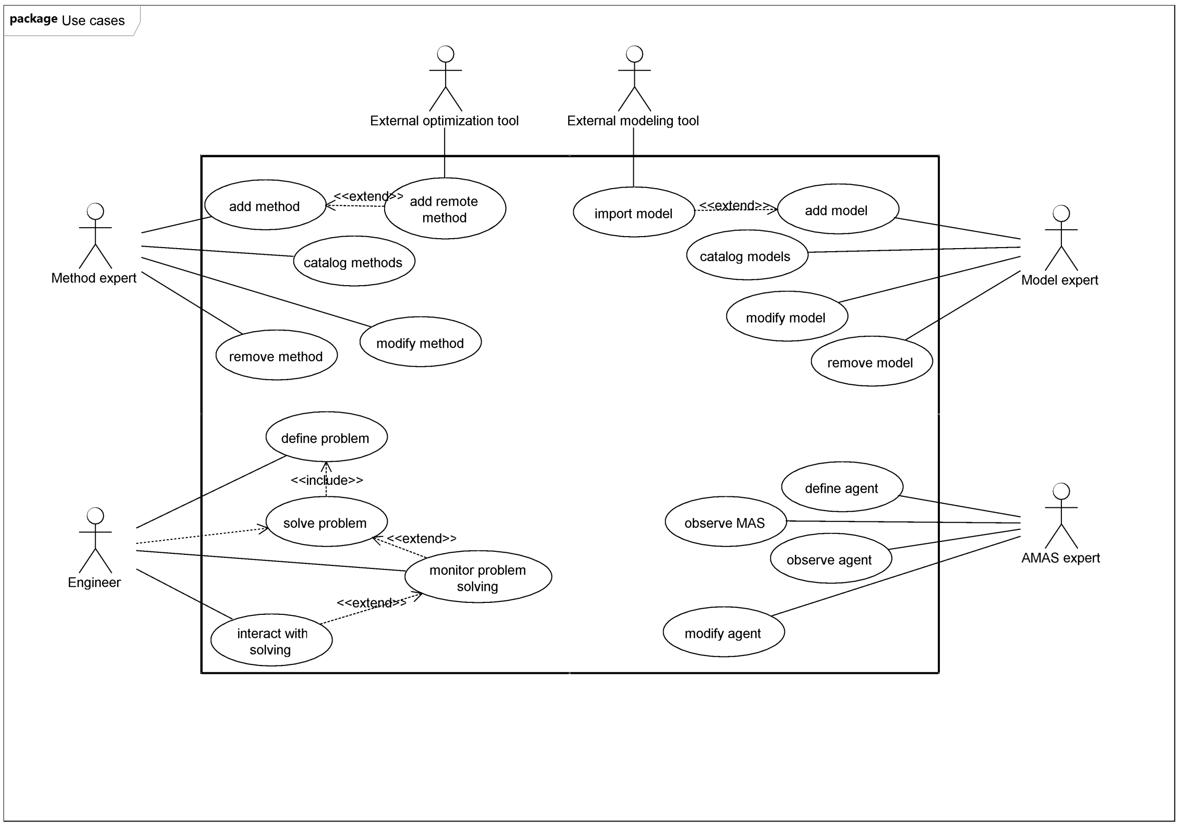


Figure 8.2: Actors and use cases identified during requirements studies for ID4CS.

Several entities interacting with the tool were also identified. The *Method expert* adds new optimization methods into the system. The expert field can be optimization methods (classic methods, AMAS methods ...) or uncertainties. The *Model expert* adds new models into the system. The role of the *Engineer* is to use the different elements defined by the other experts to define new problems for the system to solve and to monitor the functioning of the system. At last, the *AMAS expert* has to define and tune the agents of the system. The system should also be able to integrate with external optimization and modeling tool, in order to accommodate the existing workflow and constraints of the users.

The different use cases which were identified for each of these entities are presented in Figure 8.2.

The domain analysis (activity 10) roughly corresponds to the work we provided in 5.1, as this activity relates to the study of the application domain in interaction with the domain experts. The domain modeling roughly corresponds to an extended version of NDMO, with some additions and adjustments regarding the expected functionalities of the prototype. These adjustments mainly concern the introduction of the *workspace* and *project* concepts, which allow the designer to organized its work, as well as the introduction of the external optimizer and uncertainties propagators as domain entities, in order to provide a more seamless integration of these tools into the prototype.
ADELFE then advocates an AMAS adequacy verification (activity 11), in which one checks using a set of questions if AMAS are a relevant solution for the problem to solve. We discussed at length in chapter 4 the advantages of the AMAS approach. Without much surprise the results of this activity were strongly in favor of using an AMAS solution, as the problem we

aim to solve is highly distributed, is not easily solvable by known algorithms, is potentially non-linear and evolving.

The next activities concerned the design of the agents themselves, their interactions, the identification and solving of the Non Cooperative Situations. The work concerning these activities is without much doubt the most difficult, involving in the ADELFE method the participation of an "agent designer". Indeed, this part of the ADELFE method stays at a high abstraction level, and we can say without much controversy that the process of designing the agents is still highly exploratory, probably currently more related to a scientific study than to an engineering development. In this regard, our contribution presented in part II provides the expected answers to these activities.

The last WD of ADELFE corresponds to the design of the agent architecture and implementation using MAY, the component-based framework developed in complement of ADELFE. While such framework aims ultimately to make the process as seamless and automated as possible by providing reusable components, its early state implied once more some exploratory work on our part in order to produce, and in the end contribute to the MAY library with our own produced components. We will now see in the next chapter the architecture we proposed for a modular AMAS agent design.

8.3 MAY Agent Architecture

To implement the MAS, we used the Make Agents Yourself (MAY) framework [Noe12]. MAY is a component-based framework that automatically generates an implementation of an agent architecture from a given description.

The ADELFE methodology proposes an abstract agent architecture (represented in Figure 8.3), which we translated into the MAY architecture description language, SpeADL. In the context of our study, as the agents communicate and interact only by messages passing we could make some simplifications in regard of the original modeling of communications and actions capability of the agents. All our agents use the same AMAS architecture. They are differentiated by specific implementations of the components. For example, *Model* and *Variable* agents will have different implementations of the *Behavior* component.
The components of a MAY architecture are connected by *ports*. A port can be though as a service interface, listing a set of available operations. Components expose lists of available and required ports. A component required ports must be linked to available ports provided by other components, satisfying these interfaces.

An illustration of this composition mechanism is shown on Figure 8.4. Two components A and B are present. The component A exposes a port P as available, which is required and used by the component B. Consequently, this last component is now able to call the operations declared by A through this port, operations which are supposedly necessary for the good functioning of B. The way ports are used to connect otherwise independent components is in this way not too dissimilar to the role interfaces play in the context of the *dependency injection* design pattern.

For the sake of clarity, the agent architecture is separated in three views: the *behavior*, *communication* and *monitoring* views.

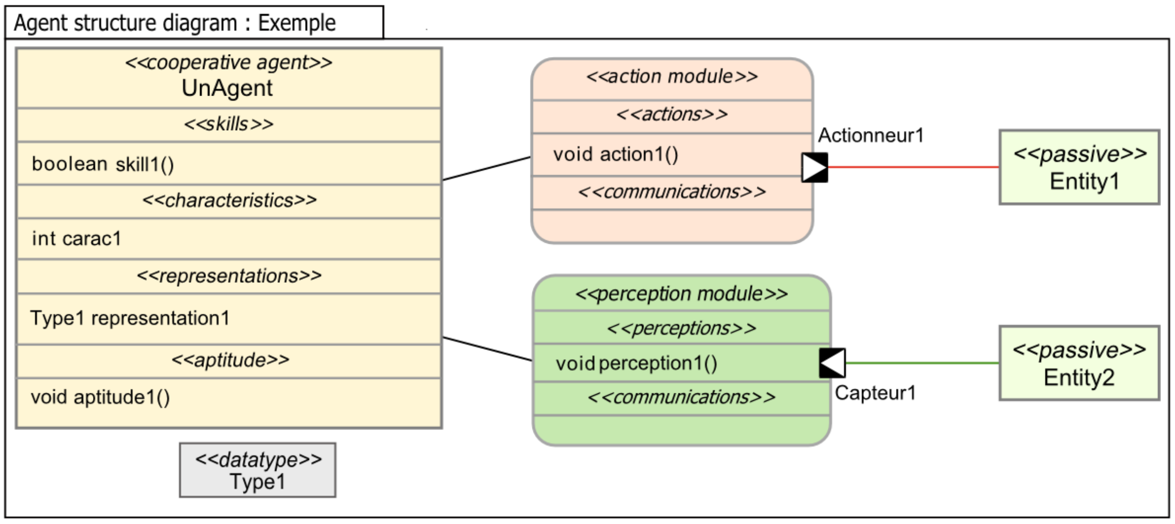


Figure 8.3: AMAS agent architecture (as defined in ADELFE).

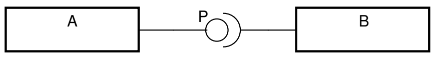


Figure 8.4: Example of MAY composition.

8.3.1 Behavior

The *behavior* view (Figure 8.5) contains the components related to the behavior of the agent.

The *Behavior* component contains the rules which dictate the behavior of the agent. This component can be seen as orchestrating the architecture. This component exposes to the outside of the environment the *Step* port, which is used to make the agent execute a step. During a step the *Behavior* component executes the agent rules. These rules will in return make use of the other components of the agent.

This behavior component contains the behavioral algorithms presented in chapter 6. This component mostly contains the parts regarding the general steps of the agent behavior and making use of the more specialized methods implemented in the *Skills* and *Aptitudes* components (for example, the exact computations involved in the detection of NCS).

The *State Vault* contains the state of the agent. It is used by the components that need to save and read some state variables. Centralizing all the states variables into one component provides several benefits. First it is easier to save and restore the state of the agent, as we just need to save the content of the vault. It is also simple to share some data between components, as long as these components have access to the vault. And it is easy to provide a view of the agent state by just reading the State Vault.
This approach has however several drawbacks. It makes the code more verbose, as we need to explicitly read the value from the vault (and possibly store it to the vault if modified). It makes more difficult to track side-effects, as it is not obvious to know which component uses which value. At last there is sadly no way to strictly enforce that components only use the State Vault for storing state values, as neither Java nor MAY can provide such guarantee.

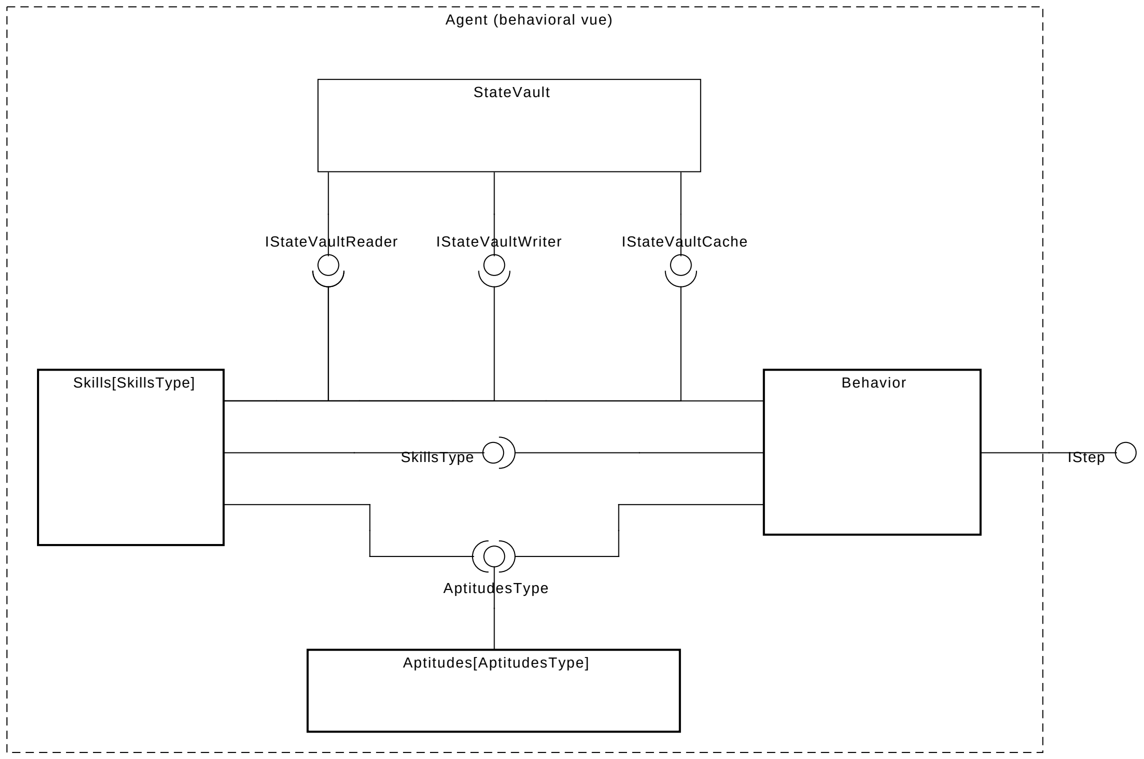


Figure 8.5: Agent Architecture — Behavior view.

The *Skills* component contains the skills of the agents. Each agent type has its own skills set, and skills can require to read and modify the agent state (thus the link between this component and the *State Vault*). Some skills are used directly from the *Behavior* rules but some skills can also be used by others skills.
Some examples of skills are: for a *Variable agent*, the capability to change its value based on the requests it received and its old value. For a model agent, the capability to evaluate an internal model and get its output values.
Skills have the somewhat unique properties that they are in themselves stackable, depending on each others. To address this concern we propose to define the *Skills* component itself as a complete components architecture in itself. We discuss this point in more details in section 8.3.1.1.

The *Aptitudes* component contains the aptitudes accessible to the agents. Unlike skills, aptitudes are general capabilities which do not rely on the state of the agent. Consequently, all agent types have access to the same aptitudes, and there is only one implementation of the *Aptitudes* component.
Some aptitudes are used directly from the *Behavior* rules but some aptitudes can also be used by skills or others aptitudes. Some examples of aptitudes are: ordering a set of requests from the most to the least important. Make some manipulations on the potentially complex exchanged values (adding, calculate the norm etc., used for example in the manipulation of uncertain values).

The distinction between *skills* and *aptitudes* is not an easy one. The ADELFE method makes the distinction between the two concepts by defining skills to be capabilities of the

agents inherent to its function, which cannot be abstracted from the application domain (for example the way a variable agent changes it value in response to the requests it receives), while aptitudes are more general "reasoning" capabilities, which could be reused between distinct systems (for example, using an Adaptive Value Tracker to track a value). The distinction between the two can be somewhat fuzzy, even more as often the firsts can depend on the seconds (for example, in the skill concerning the change of its value, the variable agent will use an AVT).
In our implementation, we choose to address this concern by classifying as skills the capabilities of the agents which are related (by using or modifying) the internal state of the agent, while classifying as aptitudes the capabilities which require no such things. Hence the distinction in our component modeling where the *Skills* component requires an access to the internal state of the agent, while this requirement is absent from the *Aptitudes* agent.

8.3.1.1 Skills Component Architecture

As stated, one of the more complex components of the agents is the one which encapsulate its skills. Indeed, each agent type has its own specific skills, and the skills in themselves can intervene at different levels or even be used by other skills. This specificity justifies the implementation of the internals of the *Skills* component itself as component architecture. As skills are, by definition, specific to the application domain this modeling concerns more the explicit handling of the dependencies between "skillsets" than the reuse of the components. However, some skills, while being specific to our application, are shared by several agent types. Based on the agent class diagram presented in Figure 5.8, we derived the skills components tree presented in Figure 8.6a. As an example, Figure 8.6b shows the composition of the *Skills* component of a model agent. As a remark, we see on the left figure that variable and output agents share the same skills. This pragmatic choice comes from the fact that, since the user can act on the problem at any time, a variable agent can become an output agent and *vice versa*.

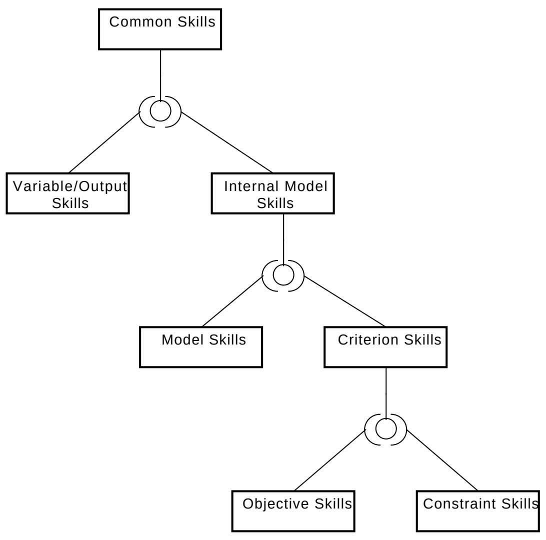
This tree-like structure of the skills components is not only a good way from engineering point of view to factor implementation code, but is also efficient in regard of the functioning of NCSs. As different NCSs are shared by different agent types at different levels, this organization enables an efficient representation of which cooperative behavior is common to which agents. The NCSs corresponding to the different components are shown in Figure 8.7

8.3.2 Communication

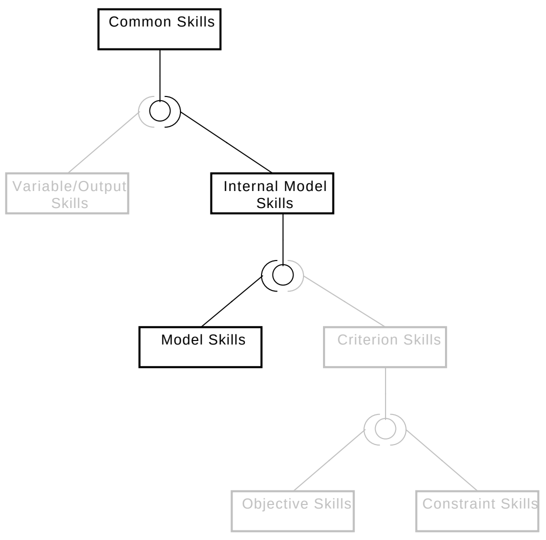
The *communication* view (Figure 8.8) presents the components related to the communication capabilities of the agent.

This view contains the *Message Box* component, which contains the messages sent to the agent. The *Message Box* stores the messages into the *State Vault* and provides a direct access to the *Skills* and Behavior components.

This figure presents several ports which need to be provided from the environment to the agent. The environment must give an unique *Reference* to the agent, which will be used by the other agents to communicate with it. The environment must also provides some ports to communicate with the other agents and outside of the system.



(a) Skills components dependencies tree.



(b) Model agent skills composition.

Figure 8.6: *Skills* component internals.

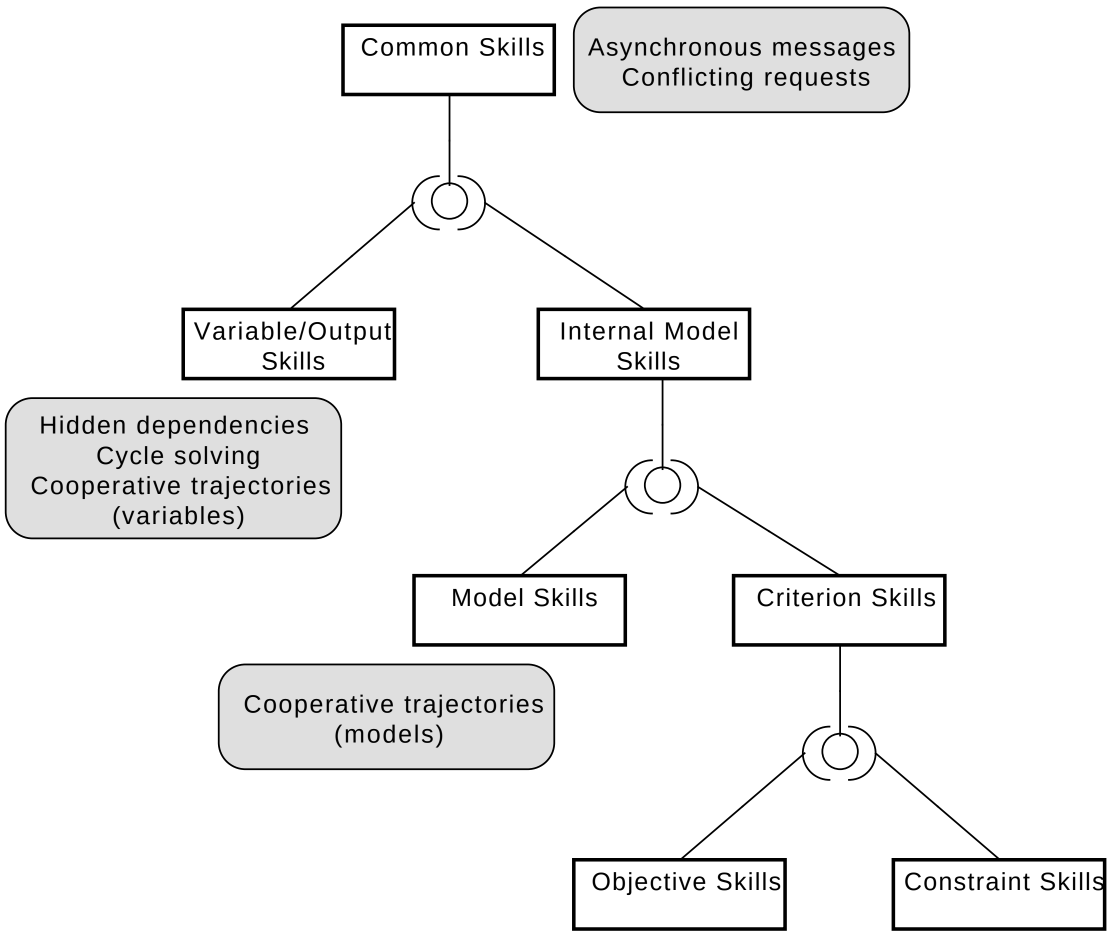


Figure 8.7: Skills components dependencies tree with corresponding NCSs.

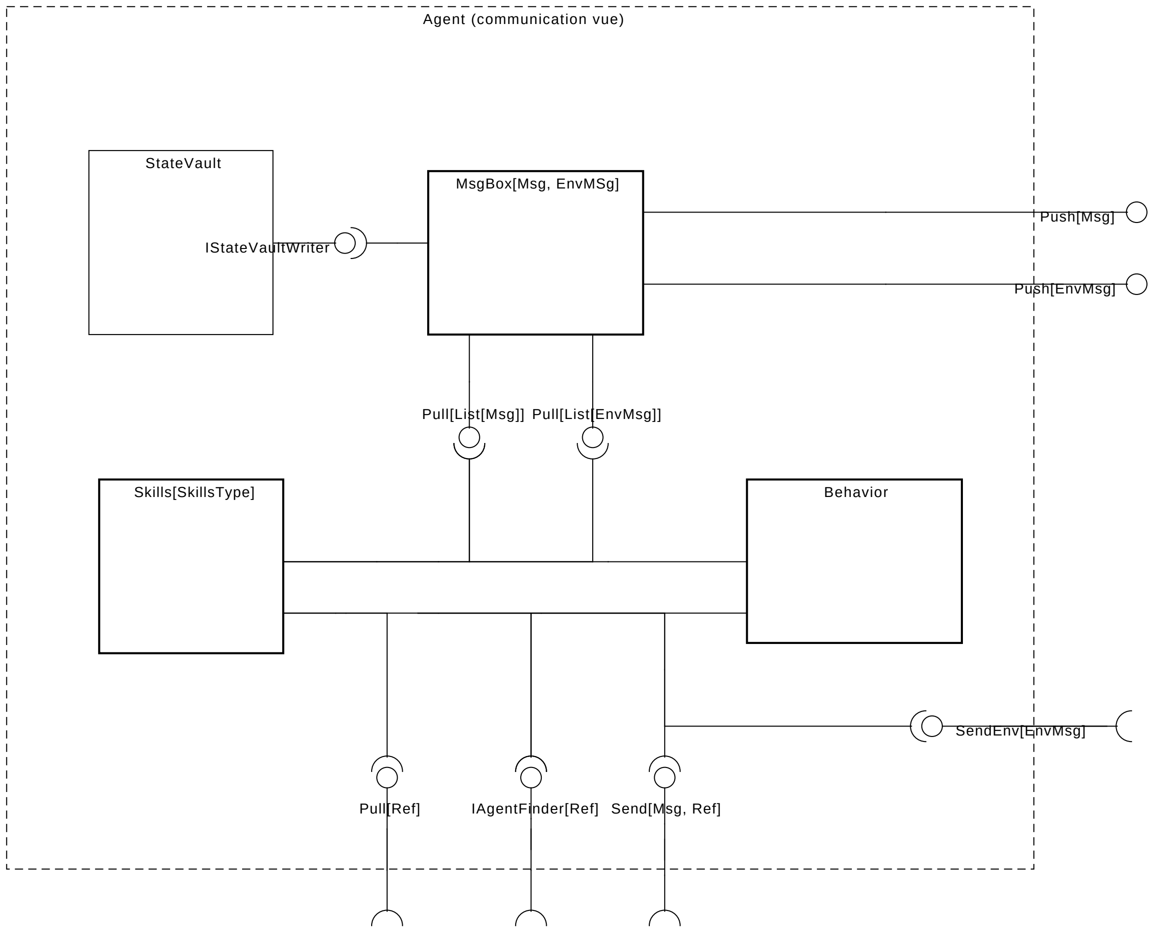


Figure 8.8: Agent Architecture — Communication view.

8.3.3 Monitoring

The *monitoring* view (Figure 8.9) presents the components related to the monitoring of the agent. It is used to observe the agents states and their modifications.

The new component introduced in this view is the *Monitor*. The *Monitor* provides two ports to the environment. The first port is used for external monitoring interfaces to subscribe to be informed of changes in the state of the agent. The second is used to provide informations concerning changes of a specific part of the agent. Thus, an external monitoring interface can subscribe in order to be notified when the state of the agent changes using the first port, and then use the second port to access to the specific information it wants to monitor.

In order to provide its capabilities, the monitor component needs to be informed by the *Behavior* component before and after each step, to read and compare the monitored informations into the *State Vault*.

8.4 MAY MAS Architecture

The MAY framework does not only allow to design the architecture of the agents, but can also be used to design the whole architecture of the MAS. This MAS architecture defines

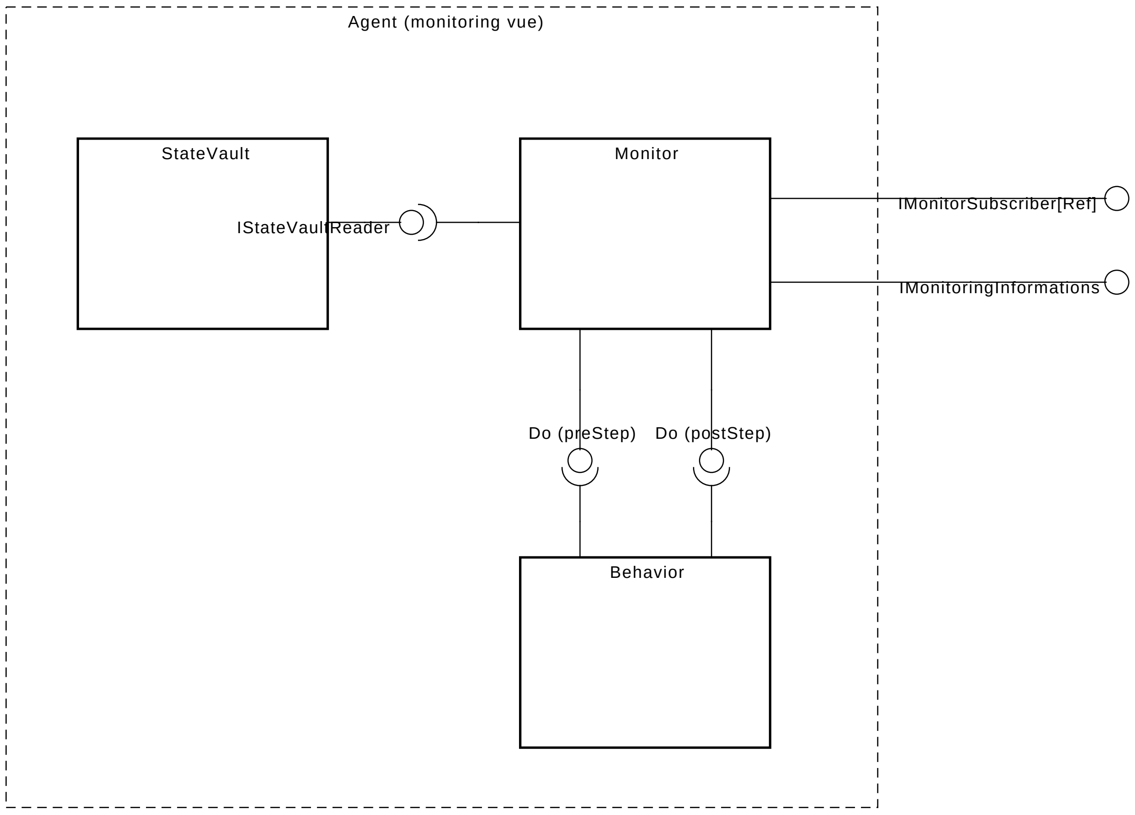


Figure 8.9: Agent Architecture — Monitoring view.

how the agents can interact among themselves and with their environment. Concerning our prototype, this architecture first provides support for message passing among the agents. The architecture of the system, which is explained below, is shown on Figure 8.10.

In order to support the integration of multiple agent instances into the global system architecture, MAY provides the concept of *transverse*. A transverse is a special kind of component which makes the liaison between a component on one side and multiple instances of a component on the other. The main use of transverses is to connect all agents instances to the other components of the system.

The components *SendService* and *ReceiveService* are MAY components used to exchange messages between the agents. While they are usually directly connected together, we inserted between them a *MsgMonitor* component, which allows us to monitor the messages transmissions. The agents also have the capability to create log messages to be written into log files using the *Log Service* component.

Some interfaces for configuring and monitoring the agents are exposed to the outside of the MAS through the *Configuration* and *Agent Monitor* transverses. The *Agent Monitor* component exposes some representative information on the agents, while the *Configuration* component offers a direct access to the agent state vault, allowing to examine the raw data as well as offering ways to the user to interact with the agents (changing values, constraints thresholds *etc.*).

A more complex monitoring interface is provided by the *System Monitor*. This component

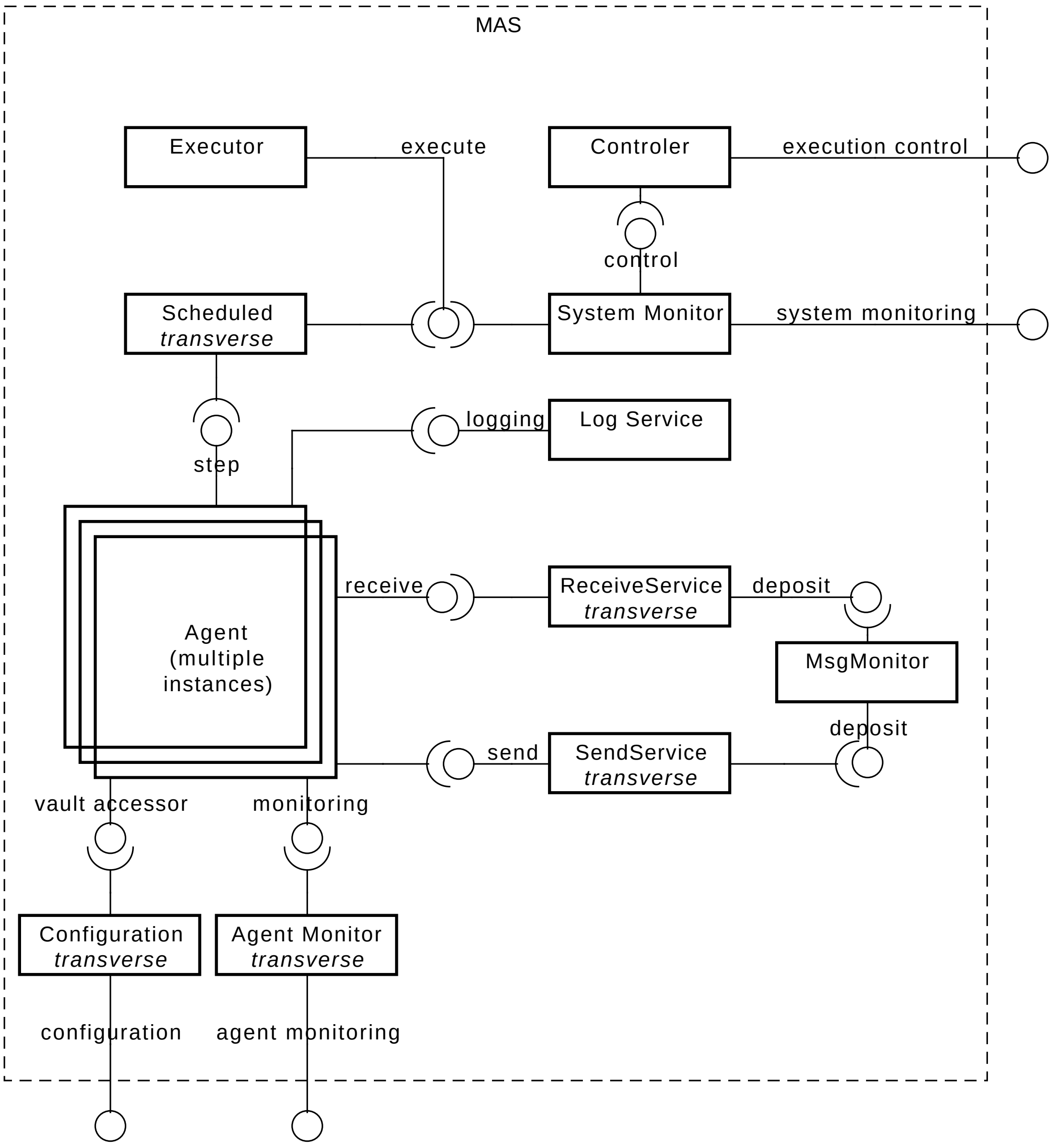


Figure 8.10: MAS Architecture.

not only allows for advanced monitoring functionalities (like subscribing to changes notifications), but also handles the execution of the system though the *Executor* component. The *Executor* handles the low-level execution concerns, like creating threads and executing tasks, and is in charge of executing the code of the agents through the *Scheduled* component. A very similar configuration (not presented on the diagram) allows the agents to exchange messages with the outside of the system. The *System Monitor* is linked to a *Controler* component, which is in charge of exposing a convenient execution control interface for the user or for an external program.

8.5 Integration into the Prototype

This implementation of the MAS was part of a collective development effort to provide a functional prototype. The goal of this prototype is to be an end-user aimed tool of the possibilities offered by agent-based continuous optimization.

The development of the prototype was carried mainly by three partners of the ID4CS project, including ourselves, each in charge of a different aspect.

The prototype can be divided in three parts: The Graphical User Interface (GUI), the core module (CORE) and the Multi-Agent System (MAS). The CORE provides a common representation of the manipulated data and is in charge of maintaining consistency between the GUI and the MAS.

These three modules communicate using the OSGi framework[2].

8.5.1 MAS

The MAS implementation is the main contribution of this thesis and was already presented in the previous parts. The only additional work was to encapsulate the implementation into an OSGi bundle which exposes the functionalities corresponding to the external services exposed by the MAS architecture and presented in the previous section.

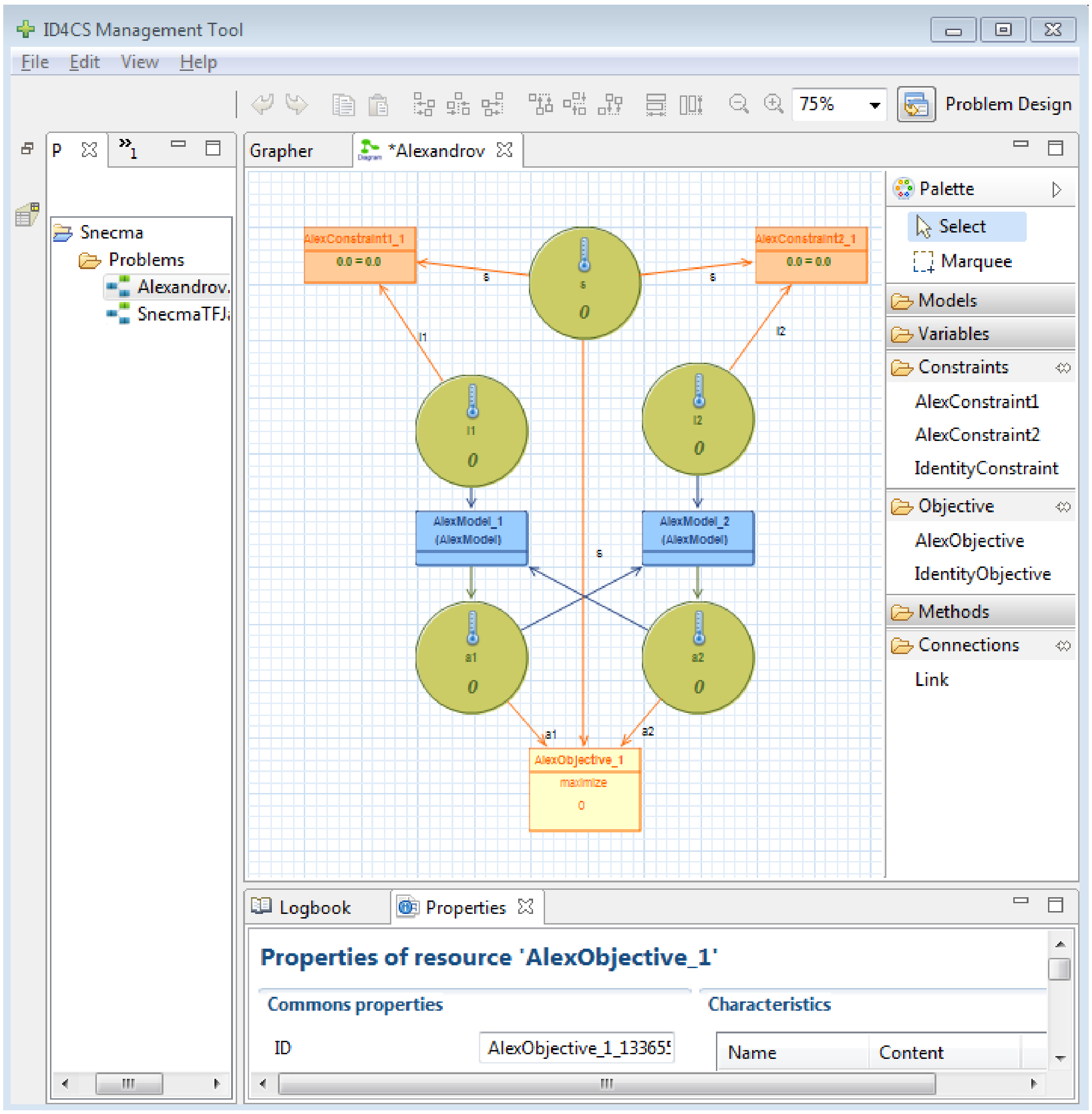
8.5.2 CORE

The CORE module is responsible of maintaining consistency of the manipulated data. It serves as a middle-man between the MAS and the GUI. It also makes possible to enable data persistence by providing a serialization/deserialization service.

8.5.3 GUI

The GUI was build using the Eclipse Rich Client Platform (RCP)[3]. RCP allows to compose graphical interface components into a user interface. It was used to propose a graphical tool inspired by existing development environments, providing the user with the possibility to define workspaces in which it can define problem elements which can then be used to define optimization problems.

[2]http://www.osgi.org
[3]http://wiki.eclipse.org/index.php/Rich_Client_Platform



Properties of resource 'AlexObjective_1'

Commons properties

ID AlexObjective_1_13365

Characteristics

Name	Content

Figure 8.11: User Interface of our prototype (problem canvas view).

By combining these three modules, we obtained a prototype which can be used to create optimization problems. To create a problem, the user can either provide a textual definition or use the graphical tools. In the last case, the user defines different reusable components (variables, models *etc.*). He is then able to drag-and-drop them from an element palette on a graph canvas and to draw links between the different elements, effectively creating the problem using our NDMO graph representation. Either way, the problem is automatically translated in an instance of the MAS, which can be controlled by the user during the optimization process. A screenshot of our prototype, showing the graph canvas on which the graph of a problem is currently being created, is shown on Figure 8.11.

9 Collective Problem Solving Patterns

We have seen in the previous chapter how the ADELFE methodology was used for the design of AMAS, from the general requirements to the implementation of the agents. However a current limitation of ADELFE (shared with most of the existing comparable methods) comes from the fact that it aims to be applicable for designing MAS intended for diverse application fields (problem solving in its broadest sense, simulation, *etc.*). This desire to be usable for a large spectrum of applications has the drawback that the recommendations and guidelines of ADELFE are often quite abstract and high-level. This makes the task of actually designing an AMAS for a precise application domain difficult for non multi-agent experts.
This limitation has already been observed in previous works, and has given birth to an ongoing effort in our team to provide a modular toolkit named *AMAS for Optimization* (AMAS4Opt), with the goal to supplement ADELFE and assist AMAS designers when designing AMAS for problem solving. In the same way that some contributions have already been proposed in the context of combinatorial optimization (see [Kad11]), we will now see how we can propose to enrich the toolkit in the context of continuous optimization.

In this chapter, we take a step back from the MAS we described in part II and see how our contribution can be made more general, not only to the benefit of the scientific community, but also for engineers by enriching AMAS4Opt.
Continuous optimization was a mostly unexplored application domain in regard to multi-agent based algorithms. By taking the (somewhat ambitious) task to propose a MAS which would be applicable for this domain as a whole, some of the patterns identified and some of the mechanisms we proposed can be used in a more general context than our system.

As continuous optimization is in itself an abstract mathematical field, we too had to abstract ourselves from concrete applications. We did not have the possibility to reduce the set of possible configurations and thus we had the occasion to encounter a variety of problems which have been mostly ignored before. Indeed, the graph representations of numerical optimization problems are quite diverse, and can present some topological properties not present in existing MAS formalisms.

In the description of our system, we presented a set of NCSs (*Non Cooperative Situations*), and the specifics mechanisms we introduced to handle them. We believe that these NCSs are only the instantiation of more general problematic topologies, which we name *Collective Problem Solving Patterns* (CPSP). The patterns are not restricted to continuous optimization and can potentially be encountered in all sorts of application domains.

Architecture and software development has greatly benefited from the identification of common design patterns. In the same way, we believe that the identification of these patterns as such, as well as of specific solutions to handle them, could lead to a great improvement for the design of agent-based systems as a whole.

Consequently, we will present in this part how the NCSs we identified during the design of our system can be abstracted in the broader form of CPSPs.

9.1 Introduction - Collective Problem Solving Patterns are not Design Patterns

Before describing the CPSPs in themselves, we must explain how these patterns differ from the existing design patterns for MAS.

There is already an existing (if limited) corpus of design patterns for MAS. These patterns have usually in scope either the design of the organizational structure of a system or the design of the behavior architecture of the agents. They concern the *design* of the system itself regarding the target application domain. Consequently they are relevant for the design of the *organization of the system*, according to the application domain. What we propose here is a different sort of patterns, which concerns the *behavior of the agent*, according to an existing organization. Design Patterns concern the structural aspects of the system, while Collective Problem Solving Patterns concerns its functional aspects.

In this regard, CPSPs are less generic than Design Patterns. Indeed the latter can be applied to the whole range of MAS, while the former only concerns MAS designed for problem solving (excluding, for example, MAS for simulation).
These two kinds of patterns should be seen as complementary. First the designer could use design pattern to design the structure of the MAS according to the needs of the application domain. Then he could use CPSPs to identify and solve specific problems resulting from such modeling and from the application domain itself.

As we want a description of our patterns which is domain-free, we cannot re-use the NDMO modeling we introduced in 5.1, which is dedicated to the domain of continuous optimization. Consequently, we will now introduce a higher-level formalism, which concentrates on the relations between the agents of the system. To keep this formalism short and simple, we will make some assumptions about the functioning of the system.

We consider systems composed of autonomous agents and resources. An agent may require that some resources be in a specific state to accomplish its local goal. We will also suppose that a resource is controlled by one agent and one agent only. We believe that this simplifying assumption does not impede on the generality of the formalism (since a system where two agents share the control of a resource can be viewed as equivalent to a system where both agents send requests to a third one solely in charge of it). At last we will suppose that agents interact among themselves by direct message passing.

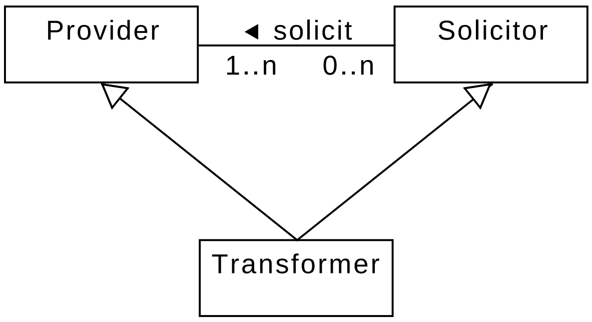


Figure 9.1: class diagram of the Provider-Solicitor modeling.

9.2 Description of a Problem Solving Pattern

9.2.1 Agent Roles

The work of [Kad11] proposed a modeling of agent roles related to the application domain of constrained combinatorial optimization composed of the *Service* and *Constrained* roles. While this taxonomy was adequate for the description of this application domain, it is not general enough for our goal. Since we want the CPSPs to be abstracted from any application domain, it is also necessary for the agent role modeling to be abstracted. Concerning the patterns we present in this article, we identified three different types of agent *roles*: Provider, Solicitor and Transformer. There is an obvious matching between *service* and *provider* roles, and between *constrainted* and *solicitor* roles, indicating that these two modelings are essentially representing similar things at different abstraction levels.

The *Provider* role represents the fact that the agent is in charge of a given resource, which can be of use to others agents in the system. The agent is responsible for choosing the state of the resource or giving access to it based on solicitations of the others agents.

The *Solicitor* role represents the fact that the agent requires that some resource(s), which it does not control, be in a specific state, in order to accomplish its goal. Consequently, the agent needs to solicit the agent(s) controlling the relevant resource(s).

The *Transformer* role is a combination of the Provider and Solicitor roles. The transformer agent controls a resource but the state of the resource is dependent of some other resources not controlled by the agent. While this role can be represented by assigning both Provider and Solicitor roles to the agent, we found this role common enough to be worth a specific representation. As we will see, transformer agents sometimes play a specific role in some CPSPs, as they can be a source of delay or obfuscation of information.

On Figure 9.1 is shown the very simple class diagram representing the relationships between these three roles.

It is important to understand that an agent is not limited to one role only. For a given system an agent can, depending on the context, assume any combination of these roles. Thus an agent can both solicit others agents regarding a resource, while being at the same time a provider of another resource. In this regard, an agent can even be a producer and solicitor of the same resource. For example the agent is in charge of a specific resource, but also benefits from it. In this case the agent can possibly be in conflict with other agents regarding the state of the resource, and decides (as a producer) to go against its own interest (as a solicitor) in order to help another agent deemed more important. Obviously, in most implementations, the different roles of the agent would not be as much segregated, and the agent would not strictly communicate with itself using message passing. This distinction should not be a problem in practice (this kind of configuration can however trigger other CPSPs, see for example 6.3.5).

For example, in the case of our system, the different agent types have relatively defined and fixed roles. A design variable agent linked with at least one other agent has a *provider* role. Constraint and objectives agents are *solicitors*. Models agents have a *transformer* role, as for output agents which are in input of at least one model or criterion agents.
The only agents which do not have roles corresponding to this taxonomy are the variable and output agents which are not used as input by any other agent. And indeed in our system these agents would have a basically non-existent role. Of course the user can still manually intervene to change the topology of the problem, changing at the same time the roles of these agents.

9.2.2 Pattern Description

For each CPSP, we provide a short description. This description will obviously be quite similar to the description of the NCSs in 6.3, as they correspond to the same pattern, only from different abstraction levels.

As we already presented the details of the different mechanisms in section 6.3, we will not detail them again in this part. Moreover, the CPSPs aim to be general patterns applicable to multiple domains, consequently it is not possible to fully specify them. A part of the instantiation work must still be done by the designer. However we discuss some conditions which are necessary in order to instantiate to the domain the handling mechanisms we proposed.

For each CPSP, we also provide a synthetic "blueprint" which is composed of two parts:

1. a representative agent configuration of the CPSP (using the agent roles introduced in section 9.2.1)
2. a summary of the mechanisms involved in the solving of the CPSP.

These blueprints are based on the template shown on Figure 9.2.

9.3 Identified Collective Problem Solving Patterns

In this section, we present the CPSPs we identified from the NCSs we encountered during the design of our MAS.

9.3.1 Conflicting Requests

In this situation a provider agent is requested to do different conflicting actions from several solicitors agents. This CPSP is probably the simplest one and was already identified as a recurring situation in previous works on AMAS, as it is the direct instantiation of the *Conflict* NCS category (as presented in section 4.2).

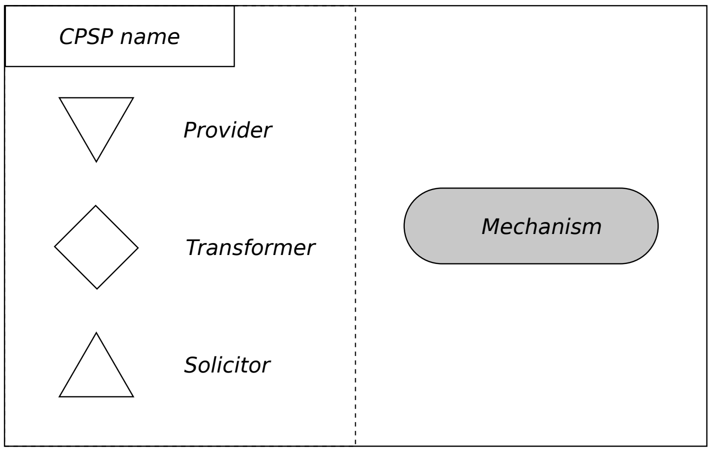


Figure 9.2: Template on CPSP blueprints.

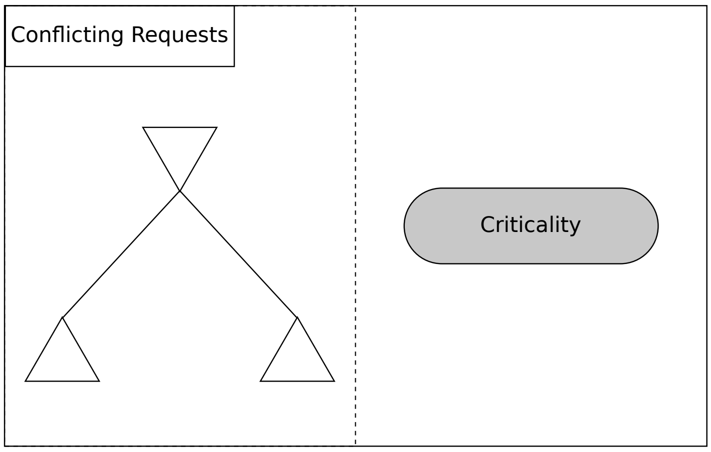


Figure 9.3: Conflicting Requests blueprint.

The blueprint of this CPSP is shown on Figure 9.3.

In order to apply the blueprint, the designer must be able to define a common and comparable measure of the "dissatisfaction" state of the solicitors. This *criticality* must be transmitted with the requests made by the solicitor agent.

The use of a criticality measure as way to discriminate between the different requests is a "tried-and-true" technique which has been applied to multiple AMAS applications[1].
As a remark, a measure of dissatisfaction presents an advantage over a measure of satisfaction in the fact that it is often possible to estimate the state of maximal satisfaction of the agent (every requirement is perfectly satisfied), but not always possible to do so for the maximal dissatisfaction. Consequently a measure of satisfaction would have an upper bound but no lower bound, while a measure of dissatisfaction has a lower bound and no upper bound. This latter is easier to manipulate and reason with as it can be easily be represented by an unbounded positive value.

[1]`http://www.irit.fr/-Projects,473-?lang=en`

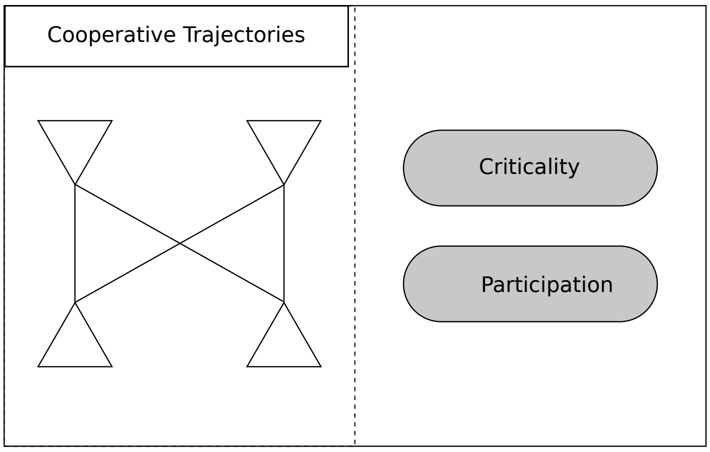


Figure 9.4: Cooperative Trajectories.

9.3.2 Cooperative Trajectories

This CPSP is an extension of the Conflicting Requests CPSP. In this case, several providers agents are solicited by conflicting solicitors. However the impact of each provider on each solicitor is different. Consequently the provider agents should be able to coordinate in order to improve every solicitors, but fail to do so because they cannot discriminate between the contradictory requests.

The blueprint of this CPSP is shown on Figure 9.4.

To apply the blueprint, the designer should introduce, in addition to a criticality measure, a participation measure representing for a solicitor the impact of a provider regarding the rest of the system.

9.3.3 Cycle Solving

This CPSP happens when several transformer agents are dependent of each others to provide their resources. It can lead to cycling behaviors where one agent sends a request to the other agent, which sends back a request to the first agent *etc.* Depending on the nature of the system and the configuration of the problem, this cycle can naturally converges toward a fixed point, or diverges, never managing to reach an equilibrium.

The blueprint of this CPSP is shown on Figure 9.5.

To apply the blueprint, the designer must add to the exchanged messages a unique signature allowing to identify without ambiguity the agent at the origin of a message.

9.3.4 Hidden Dependencies

The hidden Dependency CPSP arises when a solicitor agent assumes the agents to which it sends requests are independent providers, while one of them is in fact dependent of the other (transformer role). This pattern leads to a suboptimal behavior when the solicitor agent sends requests which are contradictory for the "top-most" provider agent.

The blueprint of this CPSP is shown on Figure 9.6.

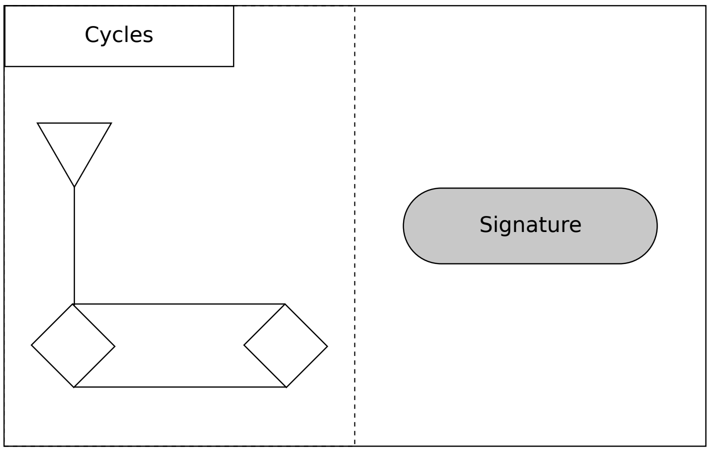


Figure 9.5: Cycle Solving blueprint.

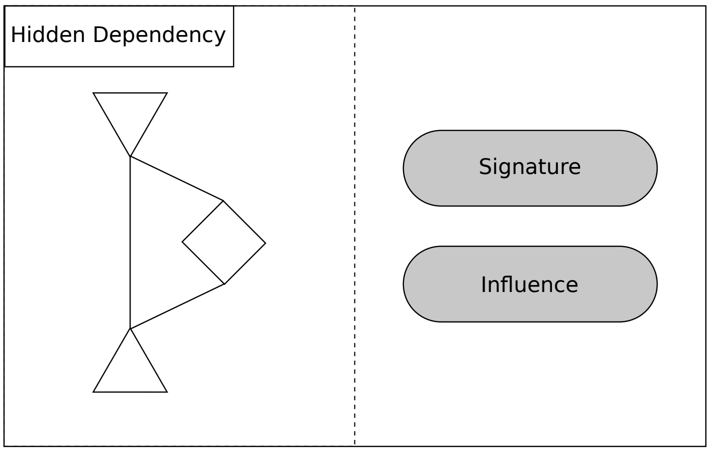


Figure 9.6: Hidden Dependencies blueprint.

To apply this blueprint, the designer has to complement the messages exchanged by the agents with unique signatures which uniquely identify the agent that is the origin of the messages. The designer must also be able to define an *influence* measure to be transmitted with the request, that represent the impact of the recipient on the solicitor. These *influences* measures must be comparable for a same origin.

9.3.5 Asynchronous Requests

This CPSP arises when a provider agent receives requests from multiple solicitor agents, but these requests arrive in a desynchronized manner. A possible suboptimal behavior can happen when the provider agent decides to satisfy a request from a solicitor, and receives afterward a more important request contradicting the first one.

The blueprint of this CPSP is shown on Figure 9.7.

To apply this blueprint, the designer must be able to define an *influence* measure for each solicitor agent, that represents the impact of the recipient on the solicitor. This *influence* will allow the agent to determine when it received enough informations to make a sufficiently informed decision, without needing to wait for *all* the messages.
Alternatively, if the delay of the agents answers is negligible, or if for any other reason it

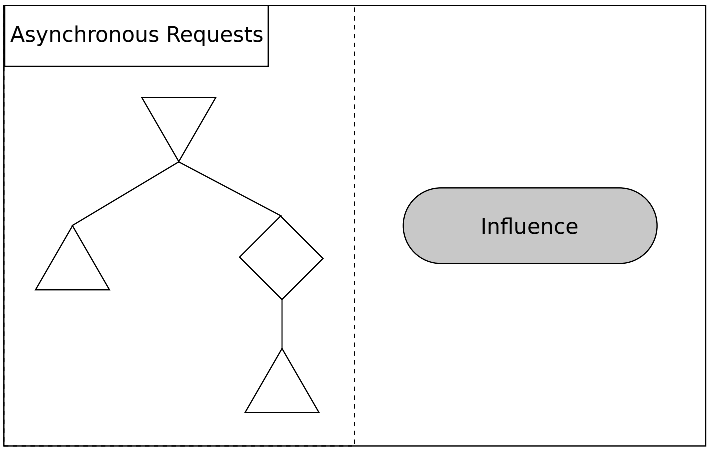


Figure 9.7: Asynchronous Requests blueprint.

is deemed acceptable that an agent waits for all the messages before taking a decision, the mechanism can be adapted to avoid using *influence* altogether.

9.4 Conclusion on Collective Problem Solving Patterns

We have seen previously that the main contribution of this thesis was the proposal of a novel agent-based algorithm for the solving of complex continuous optimization problem. In this chapter we present an additional contribution in the form of general Collective Problem Solving Patterns. These CPSPs are the generalization of the different NCSs we identified during the design of our MAS and have for goal to assist the designer of the system in the identification of potential problems which may arise from the agents organization, as well as to propose some possible handling mechanisms to solve them.

We presented a general agent role modeling which is abstracted from any precise application domain. This modeling presents the *Provider* and *Solicitor* roles, and their extension by the *Transformer* role.

Using this modeling, we presented several CPSP *blueprints* which expose in a synthetic manner the base agent pattern and the mechanisms involved in the solving of the CPSP. We also presented some of the conditions required for the designer to be able to instantiate the solving mechanism for his system.

Our agent role modeling is an extension of the one in [Kad11], in which a first abstraction work has been done to identify general agent roles for constrained optimization. This previous work led to the identification of the *service*and *constrainted* roles. However this previous modeling concentrated on the field of AMAS for constrained optimization. In this regard it was not adequate for the description of the CPSPs. Moreover, by explicitly introducing the *transformer* role in our Provider-Solicitor modeling, we are able to express more clearly some of the CPSPs (*e.g.* the Hidden Dependency pattern).
Consequently, these two modelings must not be seen as conflicting or redundant, but as complementary. The Provider-Solicitor model being more general, but more abstracted, and the Service-Constrained modeling being more specialized but more detailed in its guidelines to the designer.

As a final remark, we would like to point out how NCS-based agent behaviors, such as proposed by the AMAS theory, make a perfect fit for instantiating behavioral patterns such as CPSPs. Indeed, subsumption-based behavior architectures are very appropriate to model this kind of "exception"-like situations. Should this kind of patterns identification and reuse becomes more widely used, one could expect this way of modeling agent behavior to becomes quite popular.

An Adaptive Multi-Agent System for Self-Organizing Continuous Optimization

Experiments and Validation

In this part we present some of the results we obtained with our system on several experiments.

Our validation approach for our system was divided in three phases:

- ▷ Validating the mechanisms on simple representative test cases
- ▷ Evaluating the optimization performances and comparison with existing methods on benchmark test cases
- ▷ Testing the raw performances and scalability capabilities using automatically generated optimization problems.

The optimization problems of the first type are small enough to be solved by classical optimization techniques, however they exhibit interesting properties representative of complex optimization problems. We used these problems to identify and tune the required cooperative mechanisms for our system.
We also made additional experiments in order to evaluate others functionalities: uncertainties propagation and adaptations to changes.

The optimization problems of the second type correspond to test cases used by the scientific community as benchmarks for comparing MDO methods.

The optimization problems of the third type are algorithmically generated with the purpose of producing a base of problems of different sizes and topologies. This allows us to study the behavior when modifying the size of the problems to solve.

In every experiment, we tested the system with only the internal optimization mechanisms of the agents, without providing them with external optimization tools.

10 Behavior Validation using Academic Test Cases

This section presents some of the results we obtained on several test cases: Turbofan Problem, Viennet1, Rosenbrock's valley and Alexandrov. These test cases are not big enough to be truly qualified of "complex", but exhibit specific properties which can be found in complex test cases. Consequently they are useful to study and validate the cooperative behavior of the agents.

10.1 Turbofan Problem

We previously introduce the turbofan problem in 5.1. As stated before, the problem concerns two *design variables* pi_c and bpr. pi_c is defined inside the interval [20 - 40] and bpr inside [2 - 10]. The model produces three variables $Tdm0$, s and fr.

The problem has two objectives, maximizing $Tdm0$ and minimizing s, under the constraint $s \leq 155$ and $fr \geq 4$. This problem exhibits contradictory criteria that need to be handled at different levels ($mins$ and $s \leq 155$ needs to be handled by s, the resulting request from s and the others criteria must be handled by $TurbofanModel$), with cooperative trajectory requirements for bpr and pi_c.

On Figure 10.1, the system is executed 100 times with random starting points for each *design variable*, using only internal optimization mechanisms. As we can see, the system consistently converges toward the same optimal solution, *i.e.* the system finds an optimal values of for the two objectives with the constraints satisfied.

10.2 Viennet1

The Viennet1 test case is part of a series of problems proposed in [VFM96] to evaluate multi-criteria optimization techniques. This problem involves three objectives. Its analytical formulation is:

$$\begin{aligned}
\text{Minimize } o1 &= x^2 + (y-1)^2 \\
o2 &= x^2 + (y+1)^2 \\
o3 &= (x-1)^2 + y^2 + 2 \\
\text{where } x, y &\in [-4; 4]
\end{aligned}$$

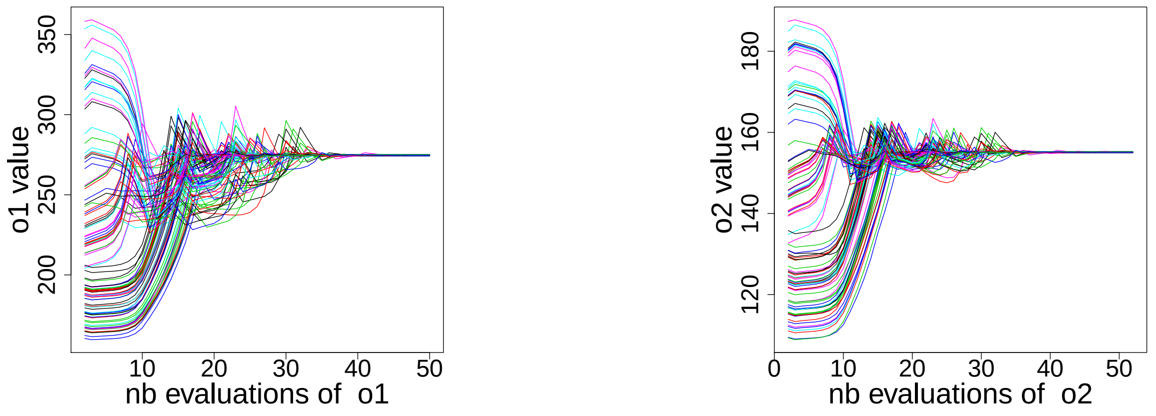


Figure 10.1: Convergence of the Turbofan objectives for 100 random starting points.

Figure 10.2 illustrates the convergence of the system toward a valid solution with 100 executions from randomly chosen starting points, using only internal optimization mechanisms.

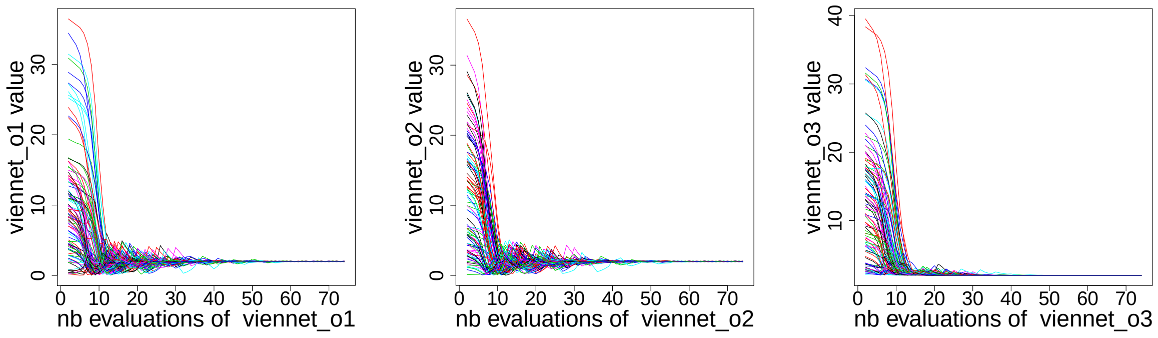


Figure 10.2: Convergence of Viennet1 objectives for 100 random starting points.

10.3 Rosenbrock's valley

Rosenbrock's valley is non-convex function commonly used to test convergence capabilities of an optimization method [Ros60].

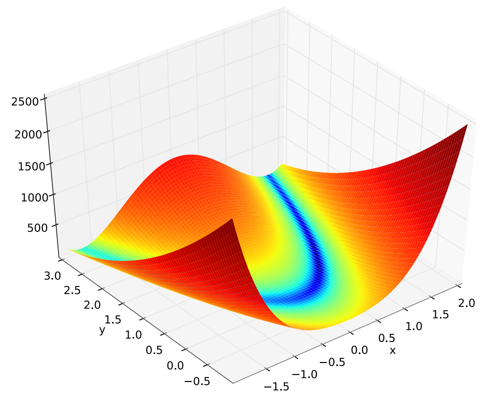
The analytical formulation of this problem (for two dimensions) is:

$$\text{Minimize } f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

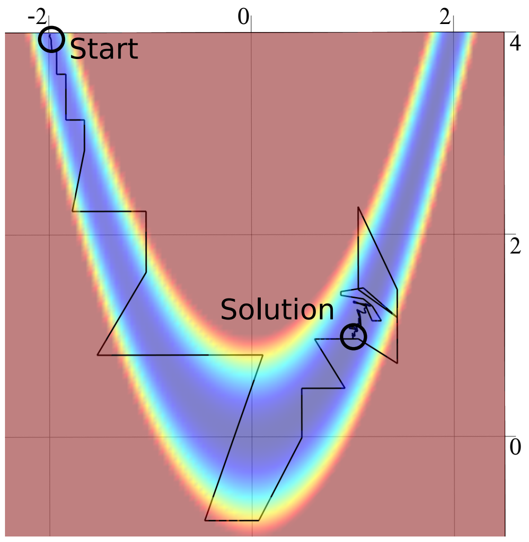
The Rosenbrock's valley problem is interesting in the fact that the global minimum is "hidden" into a narrow parabolic valley. The optimization method must thus get down into the valley and manage to follow its bottom until reaching the global optimum. Consequently it is a very adequate problem to test the cooperative trajectories mechanisms of our system.

The results presented on Figure 10.3c are for the two-dimensional version of the problem with a definition domain of [-5; 5] for each *design variable*.

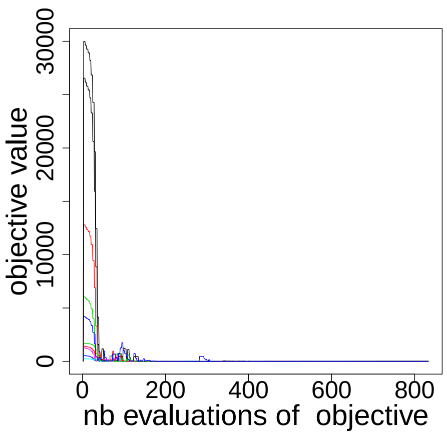
This problem was also used in [Kro+94a] as an application example of the Collaborative



(a) Rosenbrock's valley (from Martin Doege).



(b) Example of cooperative trajectories on Rosebrock's valley (starting point (-2, 4)).



(c) Convergence of Rosenbrock objective for 100 random starting points.

Figure 10.3: Rosenbrock's valley.

Optimization method, presented in chapter 3 (with performances varying from 141 to 1556 total iterations depending on the additional information used).

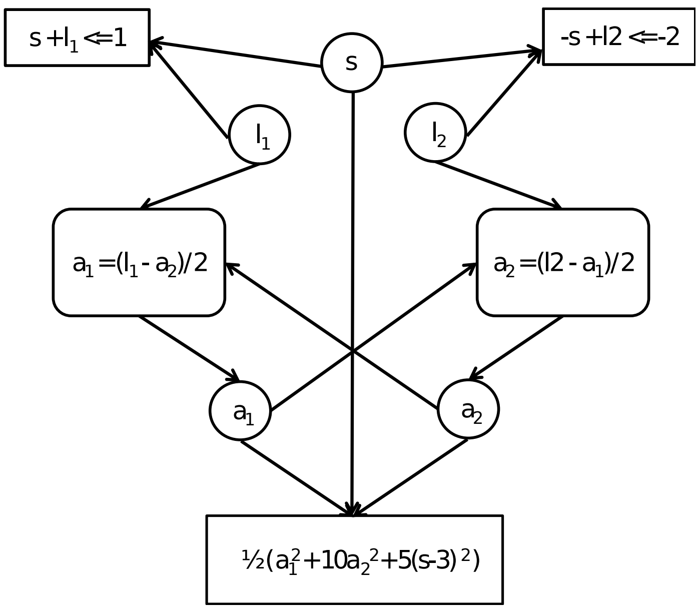
10.4 Alexandrov Problem

Our last test case is inspired from an academic example taken in literature by Alexandrov *et al*[AL02]. This example presents many of the commons characteristics of MDO problems: a cycle (albeit converging) and multiple criteria requiring cooperative trajectories. In the original study, the example was used to illustrate some properties of Collaborative Optimization, which we presented earlier, in terms of reformulation. While the original work only gave the structure of the problem, we adapted it with meaningful values and equations.

The mathematical formulation of the problem and the corresponding agent graph can be seen in Figure 10.4. Interestingly, the NDMO representation is quite similar to the one

$$\begin{gathered} a_1 = (l_1 - a_2)/2 \\ a_2 = (l_2 - a_1)/2 \\ min\ \tfrac{1}{2}(a_1^2 + 10a_2^2 + 5(s-3)^2) \\ subject\ to \\ s + l_1 \le 1 \\ -s + l_2 \le -2 \end{gathered}$$

(a) mathematical formulation.



(b) corresponding agent graph.

Figure 10.4: Alexandrov problem.

adopted by the original authors of the problem.

On Figure 10.5, the behavior of the *design variables* agents l1, l2 and s, as well the evolution of the objective, can be observed on one instance of the problem with random starting points.

On Figure 10.6, we show the evolution of the objective over 100 iterations with starting points for each *design variable* randomly drawn over the interval [-100; 100]. We can see how the system converges toward the same optimum despite the wildly different initial conditions.

10.5 Analysis of Academic Test Cases

In this section we presented several academic test cases exhibiting classical properties of complex continuous optimization problems. We shown how the system was able to consistently converge on each test case toward an optimum solution, for multiple starting points.

Summarized results illustrating the convergence of the system are presented on Table 10.1. The first group of values represents the number of evaluations which was needed for respectively 10%, 50% and 90% of the instances to find the best solution. The second group represents the average distance to the best solution (truncated at 10^{-3}) among all instances at different times (0% being the start 100% being the end of the solving in the worst case).

These results tend to validate the correctness of our approach, proving that the conjunction of the agents nominal behaviors and cooperative mechanisms allows to solve correct solution to diverse optimization problems.

We will now present the second part of our validation, by making experiment on larger test cases in order to validate the system behavior on complex problem and to compare the performances of our system with other complex optimization methods.

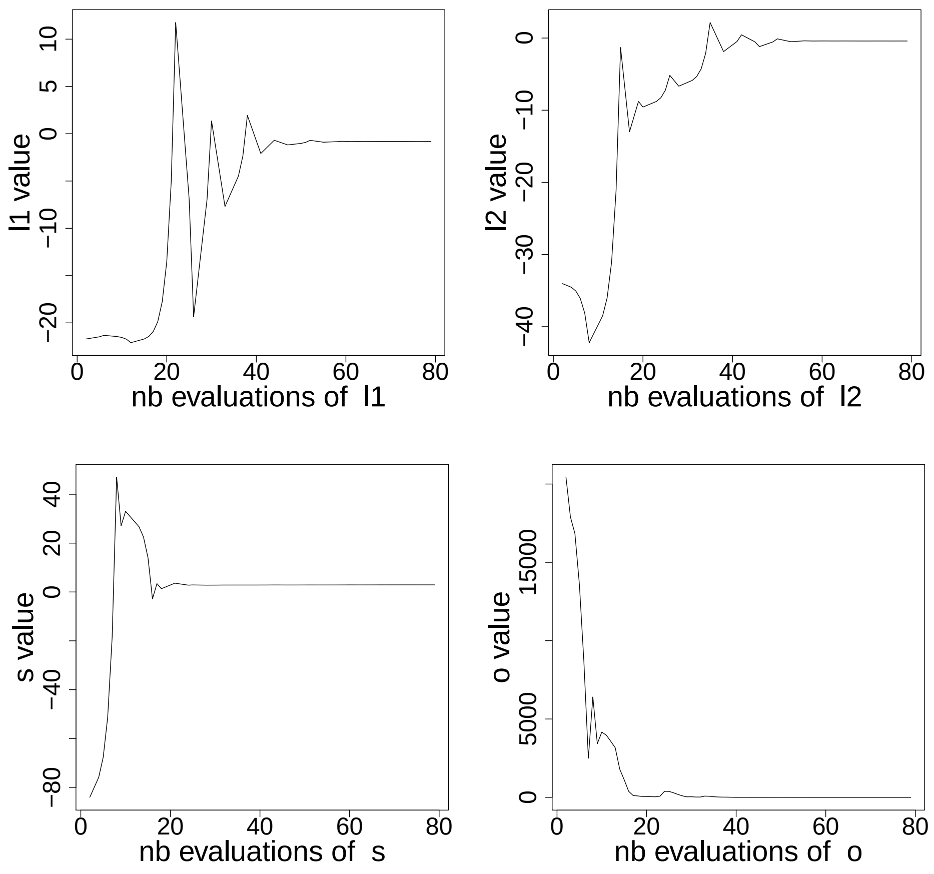


Figure 10.5: Alexandrov agents behavior.

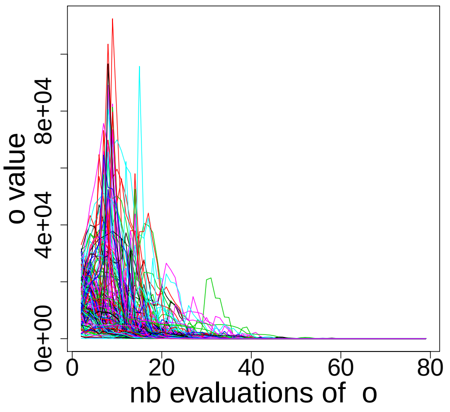


Figure 10.6: Convergence of the Alexandrov objective for 100 random starting points.

Table 10.1: Summary of experiments results for the tests cases

	nb. evaluations to best			average distance to best			
	10%	50%	90%	0% (start)	30%	60%	100% (end)
Turbofan_o1	22	38	50	67.654	21.563	1.78	0.313
Turbofan_o2	14	23	32	23.876	2.485	0.387	0.101
Viennet_o1	8	17	29	8.514	0.458	0.033	0.021
Viennet_o2	9	15	27	9.412	0.37	0.043	0.021
Viennet_o3	9	14	23	10.622	0.102	0.001	0.0
Rosenbrock	19	47	56	13749.427	123.44	4.564	2.201
Alexandrov	36	52	70	13109.169	1236.501	15.434	0.059

10.6 Optimization under Uncertainties

While it is not the focus of our system, we worked with our partners of the ID4CS project to illustrate the uncertainties propagation capabilities presented in section 7. To this end, we were provided with a preliminary aircraft design test case inspired from [SN08]. The test case is declined into two versions: a deterministic optimization problem and an optimization problem under uncertainties.

The deterministic version is the following:

$$\begin{aligned}
\min f &= \frac{2g^3 L\sigma S_d}{C_L(AM^m\sigma^n\eta_{P,Cr}E)^2\sigma(H)}\frac{1}{\left(\frac{m_{MTO}}{S_W}\right)}\left(\frac{P_{TO}}{m_{MTO}}\right)^3 \\
\text{s.t. } g_1 &= \frac{m_{MTO}}{S_W} - \frac{k_L\sigma C_{L,max,L}S_{LFL}}{m_{ML}/m_{MTO}} \leq 0 \\
g_2 &= \frac{k_{TO}Vg}{s_{TOFL}\sigma C_{L,max,TO}\eta_{P,TO}} - \frac{P_{TO}/m_{MTO}}{m_{MTO}/S_W} \leq 0 \\
g_3 &= V_{CR}\frac{V_{md}}{E\eta_{P,CR}AM^m\sigma^n}\sqrt{\frac{\pi Ae\rho_0\sigma(H)g}{4E_{max}}} - \frac{P_{TO}}{M_{MTO}}\sqrt{\frac{m_{MTO}}{S_W}} \leq 0
\end{aligned}$$

where $\frac{m_{MTO}}{S_W}$ and $\frac{P_{TO}}{m_{MTO}}$ are the two design variables of the problem, representing respectively the power/mass ratio and the wing loading of the aircraft, and the other values are constant. The three constraints concern respectively the takeoff length, landing length and cruising speed. The solution to the deterministic problem is $\frac{m_{MTO}}{S_W} = 377$ and $\frac{P_{TO}}{m_{MTO}} = 187$, for $f = 2.15E + 08$.

In the version with uncertainties, normal distribution laws are associated with some variables of the problem. The variables and associated uncertainties are shown in Table 10.2

The new objective is not to minimize f but to minimize $E(f)$, the expected value of the function. The new constraints are $P(g_i \leq 0) \geq 0.9,\ \forall i \in \{1,2,3\}$.

Table 10.2: Uncertainties associated with the variables

Variable	Expected value	Standard deviation
$C_{L,max,L}$	2.50	0.250
$C_{L,max,TO}$	2.10	0.210
E	12.49	1.249
$\eta_{P,CR}$	0.86	0.086

We slightly adapted our MAS to mimic a classical sequential optimization scheme. The basic idea of this type of method is to start by doing a deterministic optimization of the problem and, when the optimization has converged, to switch to an optimization under uncertainties using the deterministic solution point as a starting point. For our MAS, we instantiated this method by making the agents start doing a deterministic optimization. When a certain convergence condition is detected, the agents automatically switch to optimization under uncertainties. A difference is that the system is not interrupted between the deterministic optimization and optimization under uncertainties. The agents handle the transition automatically by switching to uncertain values.

The solution to the problem under uncertainties is $\frac{m_{MTO}}{S_W} = 329$ and $\frac{P_{TO}}{m_{MTO}} = 244$

For the uncertainties propagations, the agents are provided with Monte-Carlo propagators. These propagators draw 50000 random values on the inputs following the uncertainties associated with each input, and return a set of points for each outputs. The constraints and objectives can directly use these sets of points, as explained in chapter 7.

The trajectory of the system is shown on Figure 10.7. The red dashed curves represent the deterministic constraints and the red dotted curves an approximation of the constraints with uncertainties. The blue and green circles represent respectively the deterministic and robust solution points. The starting point (300, 300) is indicated by the letter S. We can see on the figure how, in the first part, the system converges toward the deterministic solution point and, on the second part, how it switches to uncertainties and changes its direction to converges on the robust solution point.

10.7 Adaptation to Perturbations

10.7.1 Perturbated Alexandrov Problem

On Figure 10.8, we can observe the reaction of the multi-agent system to a perturbation. During the solving of the previous experimentation on the problem, we changed the threshold of the constraint $s + l_1 \leq 1$ to $s + l_1 \leq -4$ (the change is indicated by a dotted line on the charts). The system dynamically adapts to the constraint changed and converges toward a new solution which satisfies the updated constraint.

10.7.2 Perturbated Turbofan Problem

On Figure 10.9, we illustrate the adaptation capabilities of the system by subjecting turbofan problem we introduced previously to a series of both strong and faster successive

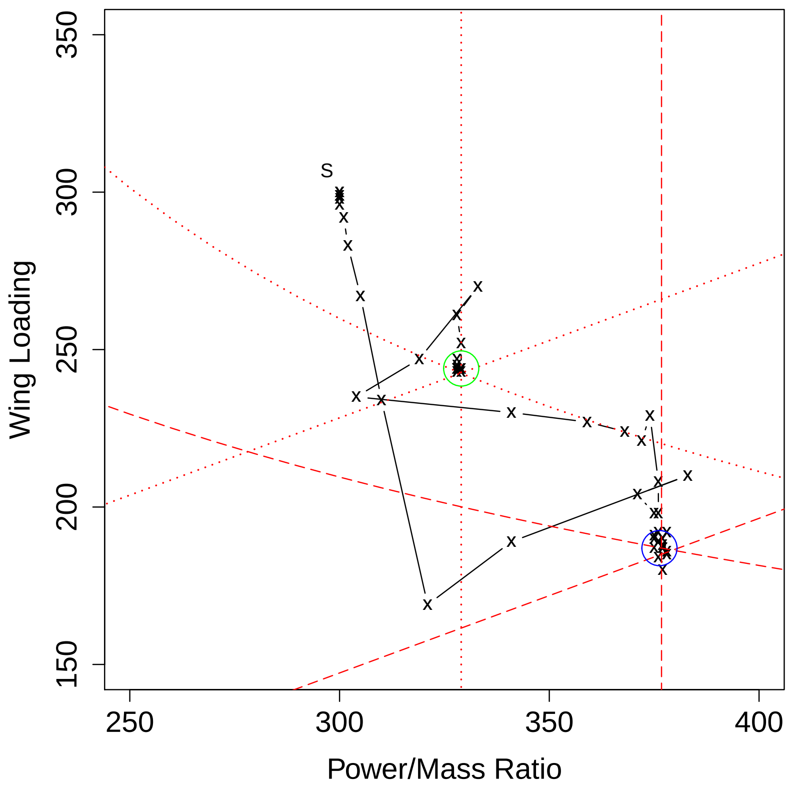


Figure 10.7: Sequential optimization trajectory.

small changes to the problem topology (each perturbation is indicated by a dotted line). First we create strong changes by modifying simultaneously both a constraint and the definition domain of *pi_c*:

a. $c1$ changed from $s <= 155$ to $s <= 165$, max bound of *pi_c* changed from 40 to 50

b. $c1$ changed from $s <= 165$ to $s <= 145$, max bound of *pi_c* changed from 50 to 30

Then milder perturbations by only changing the definition domain of the variable:

c. max bound of *pi_c* changed from 30 to 35

d. max bound of *pi_c* changed from 35 to 40

e. max bound of *pi_c* changed from 40 to 45

The experiments show that the system consistently reacts to these perturbations by adapting itself in order to find a new solution for the modified problem.

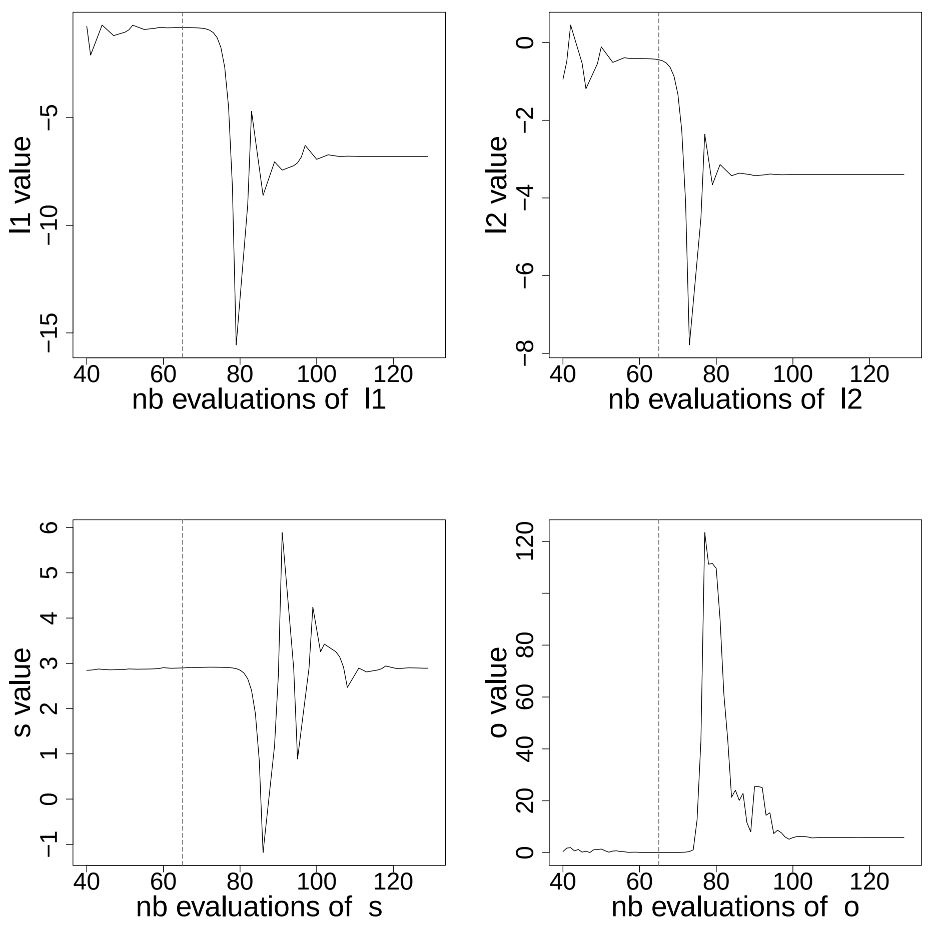


Figure 10.8: Alexandrov agents behavior with perturbation (constraint change at dotted line).

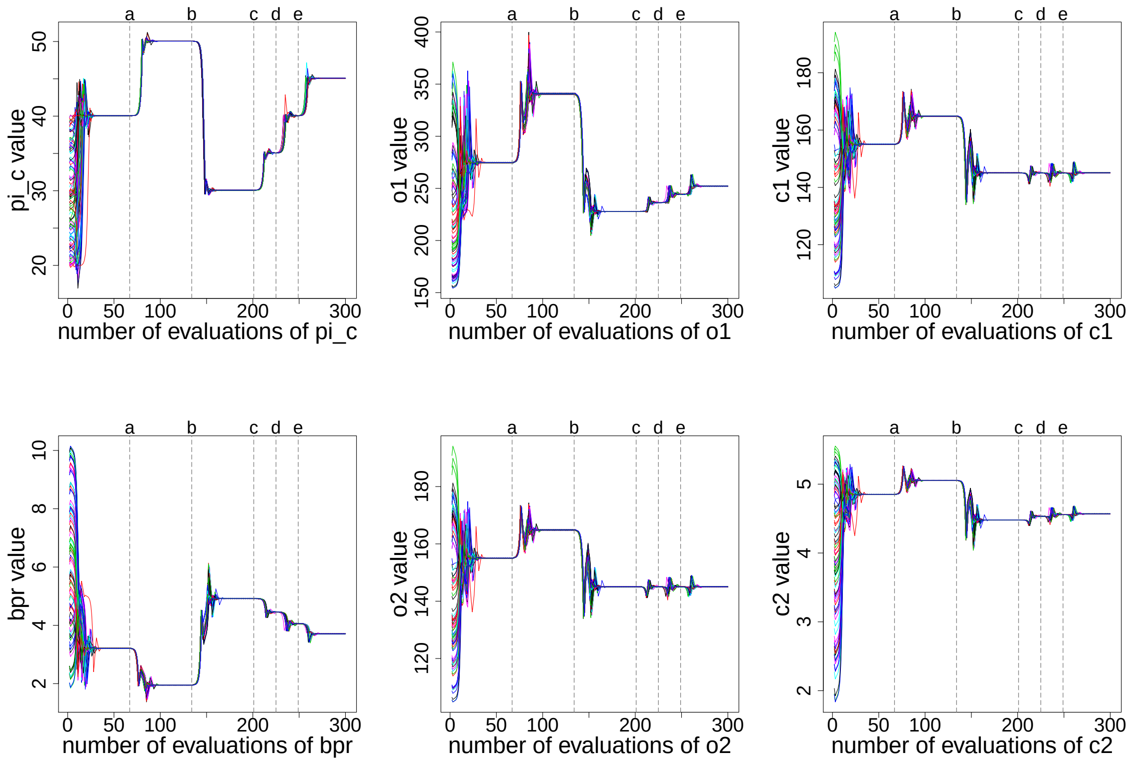


Figure 10.9: Turbofan agents behavior with perturbations (changes at dotted lines).

11 Comparison with Existing Methods

In this chapter we provide a comparison of our MAS with existing MDO methods. This comparison is based on the work of [PLB04; YSP08]. These two works are complementary as both of them concern a common subset of representative MDO methods: Multidisciplineary Feasible (MDF), Individual Discipline Feasible (IDF), Concurrent Subspace Optimization (CSSO), Bilevel Integrated System Synthesis (BLISS) and Collaborative Optimization (CO).

The work of [PLB04] presents an analysis of these methods based on two test cases and rank them on multiple criteria: *Accuracy, Efficiency, Transparency, Simplicity, Portability*.

The analysis of [YSP08] is based on different criteria, concentrating on the performances regarding the number of functions calls and the additional informations required by each methods. An interesting aspect of this comparison is that it provides several mathematical test cases on which the methods are applied.

For our evaluation we use test cases provided by both works and compare the performances of our system to the existing MDO methods. We then provide a comparison synthesis based on the criteria and analysis presented in [PLB04].

In order to obtain comparable results with MDO methods, we adopted a problem modeling based on the disciplines division proposed in these works. To this end, we considered that, for each test case, the natural formulation of the problem matched the proposed disciplines division. Consequently, for each test case, the mathematical models represented by our model agents correspond to the proposed disciplines.

11.1 Comparison Criteria

For our comparisons we use the criteria proposed in [PLB04], which are defined as follow:

- *Accuracy*: the quality of the solution proposed by the method, based on the distance between the proposed solution and the real optimal values.
- *Efficiency*: the computational cost to find the proposed solution, based on the number of disciplinary evaluations.
- *Simplicity*: the ease to instantiate and apply the method to different problems, based on the number of optimizers and variables required to implement the examples.
- *Transparency*: the capability to understand, modify or extend the method.
- *Portability*: the feasibility to apply the method in the context of an existing work

organization, based on the distributivity capabilities of the method.

The *Accuracy* criterion reflects the aptitude of the method to find the input values corresponding to the optimum of the problem.

The *Efficiency* criterion concerns the computational cost required by the method to provide its solution. When counting the number of evaluation calls, a distinction is made between discipline evaluations and full problem evaluations, the latter being considered considerably more costly than the former. Our MAS does not require any full problem evaluation, using only disciplines evaluations.

The *Simplicity* criterion concerns the amount of work required to instantiate the method to suit different optimization problem. The original authors chose to evaluate this criterion using two measures: the number of optimizers involved and the number of additional variables required by the method.

The *Transparency* criterion is the less well-defined and can be perceived as somewhat subjective. The original authors give as an example that "a probability-based method can be seamlessly integrated into a transparent formulation, which does not require major changes of the architecture to accomplish the integration."

The *Portability* criterion concerns the feasibility to integrate the method into an existing organizational structure. This evaluation criterion is based on the capabilities of the method to divide the concerns to match existing expert teams or specializations.

11.2 Comparison Problem 1

Our first comparison test case comes from [PLB04]. It was originally introduced in [SBR96] to study the performances of the CSSO method. Its formulation is the following:

$$
\begin{aligned}
\min f &= x_2^2 + x_3 + y_1 + e^{-y_2} \\
\text{s.t. } g_1 &= \frac{y_1}{3.16} - 1 \geq 0 \\
g_2 &= 1 - \frac{y_2}{24} \geq 0 \\
\text{where } y_1 &= x_1^2 + x_2 + x_3 - 0.2y_2 \\
y_2 &= \sqrt{y_1} + x_1 + x_3 \\
\text{with } &-10 \leq x_1 \leq 10 \\
0 &\leq x_2, x_3 \leq 10
\end{aligned}
$$

The problem is divided into two disciplines. The discipline 1 corresponds to the computation of y_1 and g_1, while the discipline 2 corresponds to the computation of y_2 and g_2. The authors use the same initial points $x_1 = 1$, $x_2 = 5$, $x_3 = 2$ for each analysis. The optimum for this problem is located at $x_1 = 1.9776$, $x_2 = 0$, $x_3 = 0$, for which the value of f is 3.1834.

We present here the comparison of our method with the ones evaluated in [PLB04]. We reproduce their results here and add our method to the comparisons under the term **AMAS**.

Regarding the simplicity criterion, the relevant measures are shown in Table 11.1.

Table 11.1: Comparison Problem 1 – Simplicity

Method	No. Optimizers	Additional variables
MDF	1	0
IDF	1	2
CSSO	3	3
CO	3	9
BLISS	3	3
AMAS	2	0

Table 11.2: Comparison Problem 1 – Efficiency

Method	Coordination evaluations	Discipline 1 Evaluations	Discipline 2 Evaluations
MDF	24	216	216
IDF	62	54	54
CSSO	20	528	528
CO	249	6106	4515
BLISS	40	95	95
AMAS	0	201	201

In regard of the transparency criterion, the authors in [PLB04] note that MDF and IDF are the most transparent, as the mathematical models and objectives are easily formulated and modified. The CO method requires a little more attention in the formulation of the disciplinary objectives. CSSO and BLISS are the more complex, needing respectively approximation models and a sensitivity analysis.
Our method, as MDF can be directly derived from the analytical expression of the problem, without requiring further work from the expert. However, as the problem is divided in two disciplines, it will require two optimizers.

Concerning the portability criterion, the authors note that MDF and IDF, being centralized methods, are not flexible and cannot adapt to existing organizational structures. CO, CSSO and BLISS on the other hand can be adapted to existing organization or distributed computing requirements. However CSSO requires not only for each discipline to provide an approximation of their state but need also an entire system analysis at each iteration. BLISS requires additional calculations to calculate the sensitivities of the disciplines.
Our method is similar to CO, CSSO and BLISS as the experts can choose how to divide the problem to match the organization of the disciplinary groups, without imposing additional requirements concerning the formulation of the problem. It is also naturally distributable, as each agent is an independent process.

In regard of the efficiency, the number of evaluations for each methods is detailed in Table 11.2.

Results for the accuracy are shown on Table 11.3. While MDF and IDF find the correct solution, CSSO, CO and BLISS present slight errors. The authors note that CO and CSSO introduce discrepancies in the computation of the outputs due to the approximation these methods make.

Note that in our case, the rounding is done at a lower precision than in the other test

Table 11.3: Comparison Problem 1 – Accuracy

Method	x_1	x_2	x_3	y_1	y_2	f
MDF	1.9776	0	0	3.16	3.7553	3.1834
IDF	1.9776	0	0	3.16	3.7553	3.1834
CSSO	1.9778	0	0	3.16	3.7675	3.1831
CO	1.9776	0	0	3.16	3.7556	3.1835
BLISS	1.9770	0	0	3.15	3.7544	3.1804
AMAS	1.97764	0	0	3.16	3.75528	3.18339

cases. This is voluntary to illustrate the fact that the constraint y_2 is actually violated, but this violation is lower than the rounding decimal, as per the criticality mechanism of constraint agents we explained in 6.3.1.

11.3 Comparison Problem 2

For our second comparison test case, we selected one of the problems presented in [YSP08][1]:

$$\begin{aligned}
\min f &= (z_1 - 0.5)^2 + (z_2 - 0.5)^2 \\
\text{s.t. } g_1 &= 1.0 - z_1 \leq 0 \\
g_2 &= 1.0 - z_2 \leq 0 \\
\text{where } z_1 &= (b_1 - 2.5) + (b_c - 2.0) - 0.5z_2 \\
z_2 &= (b_2 - 3.0) + (b_c - 2.0) - 0.7z_1
\end{aligned}$$

The problem is divided into two disciplines, the first one corresponding to the computation of z_1 and g_1, while the second one corresponds to the computation of z_2 and g_2.

We did not include MDOIS in the comparisons, as it is a specialized method applicable only to specific problem topologies, and not a general MDO method (see section 3.2.5). On Table 11.4 is shown the number of design variables and additional equality constraint for each method.

Sadly, the authors did not provide the starting points they used for evaluating the number of evaluations of the different methods. Consequently we cannot provide a meaningful comparison of our method on this criterion. For our experiments, we choose to make 100 instances of the problem, with random starting value for the variables drawn between -100 and 100. As an indication, the median number of evaluations for the disciplines required by our method to find the solution is indicated in Table 11.5. In Table 11.6 we show the median value obtained over all the experiments for f.

[1]the original authors originally introduced two additional variables z_1^{nc} and z_2^{nc}. However these variables had no impact on the problem in itself, consequently we did not reproduce them here

[2]results are median values over 100 iterations

[3]results is median value obtained over 100 iterations

Table 11.4: Comparison Problem 2 – Simplicity

Method	No. Design Variables	No. Equality Constraints
MDF	3	0
IDF	5	2
AAO	7	4
CSSO	7	2
BLISS	3	0
CO	9	2
AMAS	3	0

Table 11.5: Results on Comparison Problem 2 – Efficiency

Method	Coordination evaluations	Discipline 1 Evaluations	Discipline 2 Evaluations
MDF	292	0	584
IDF	0	50	50
AAO	0	178	178
CSSO	449	1096	445
BLISS	358	752	744
CO	0	969	948
AMAS[2]	0	709	709

11.4 Comparison Synthesis

Here we present a synthesis of the comparison between our system and existing MDO methods.

On Table 11.7, we reproduce the analysis of [PLB04] and add our own method (indicated as **AMAS**) to the comparison. Note that the classification of the existing MDO methods is based on the original work.

In regard of the accuracy criterion, our method provides good results, consistently converging toward the optimum in both benchmarks. We classed it under MDF, as our method still need to be provided with a precision, and allow for constraint violation under this precision.

In regard of the efficiency criterion, our method is on part with the most efficient MDO methods.Moreover, our experiments were done using only the agents internal optimization mechanisms. Even better results could be expected if the agents were provided with adequate external optimizers.

In regard of the simplicity criterion, for the number of objectives, our method requires n optimizers (n being the number of models involved). In this regard it is less efficient than MDF and IDF, which only require 1 optimizer, but is better than the other methods which require $n + 1$ optimizers (1 by model plus 1 global optimizer). For the number of additional variables, our method is on par with MDF, the best method in regard of this measure, as neither of them require any additional variable.
The only method dominating our own in regard of this criterion is MDF. Our method can be considered on par with IDF, as neither of them dominate the other on both measures.

Table 11.6: Results on Comparison Problem 2 – Accuracy

Method	f value
MDF	0.50001
IDF	0.49999
AAO	0.49957
CSSO	0.50992
BLISS	0.50105
CO	0.49223
AMAS[3]	0.50001

Table 11.7: Classification by criteria (based on [PLB04])

	Accuracy	Efficiency	Transparency	Simplicity	Portability
best	MDF	IDF	**AMAS**	MDF	**AMAS**
$\downarrow$	**AMAS**	BLISS, **AMAS**	MDF	IDF, **AMAS**	CO
	IDF		IDF		CSSO
	BLISS	CSSO	CO	CO	BLISS
	CO	CO	CSSO	CSSO	IDF
worst	CSSO	MDF	BLISS	BLISS	MDF

In regard of the transparency criterion, we have shown that our approach was modular and extensible, due to the distribution of roles between the agents and to the explicitly encapsulation of mathematical tools (analytical models, external optimizers). We demonstrated this modularity with the addition of uncertainty propagation mechanisms into the MAS.

In regard of the portability criterion, our method is extremely flexible as it supports multiple levels of modeling and multiple reformulations of the problem. Consequently the problem can be "tailored" to fit the existing organizational structure. Contrary to the MDO methods, our method does not require a *central* authority as the responsibility to maintaining consistency is distributed in the system.

12 Evaluating Scalability Performances using Generated Test Cases

12.1 Generated Problem Graphs

In this section we present some measures we established in regard of the scalability performances of our MAS. For this part of the experiments we were not concerned with the optimization performances of the system, but more with its capability to handle large agent graphs without suffering from performances degradation.

12.1.1 Generating NDMO Agent Graphs

In order to realize our experiments, we required a large number of test cases of several sizes and comparable complexity. Since such repository of continuous optimization problems is not readily available, we worked on a way to automatically generate them. To this end, we took advantage of the fact that, as demonstrated with our NDMO modeling, an optimization problem can be represented as a graph. Consequently it is possible to use well-known graph generation techniques and directly generate problem graphs.

We propose a very simple method to generate simple problem graphs. First we use a graph generation algorithm to generate a directed graph of size n. The nodes of this graph represent the variables of the problem and the arcs their dependencies. If a node is not the head of any arc (*i.e.* there is no arc going to this node), it is a design variable, otherwise it is an output variable.
The next step is to insert models. As we stated, the optimization problem in itself is of little importance in this part. Consequently our only requirement for the model is that they must be able to take an arbitrary number of inputs and produce one output. In our experiments we used a simple *Sum* function. For each output variable, we insert a node representing the model in the graph. The links going to the output variable node are redirected to the model node, and a link going from the model node to the output node is created.
After this step, criteria are randomly added to variable nodes. In our experiments, we given a 20% probability of a variable node to be linked to a criterion, with half the chance for the criterion to be an objective and half the change for it to be a constraint. Once more, in order to simplify the problem generation, all objectives are about minimizing the variable, and all constraints are about having the variable higher of equal to zero.

The generation method is summarized in 12.1. After all these steps, the graph is a valid NDMO graph and can be directly transformed into an agent graph.

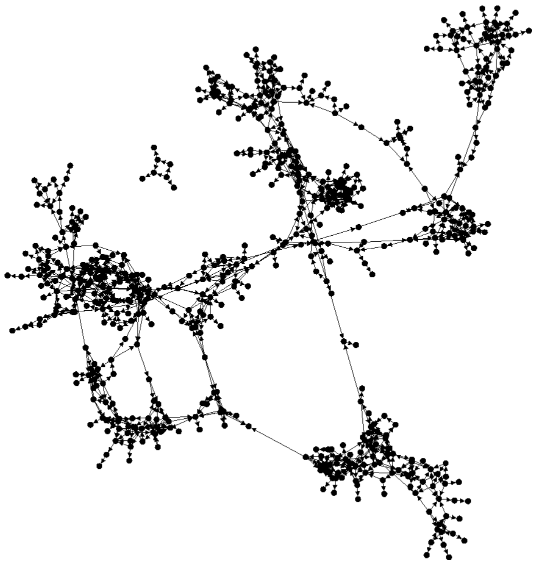
Algorithm 12.1: Problem graph generation

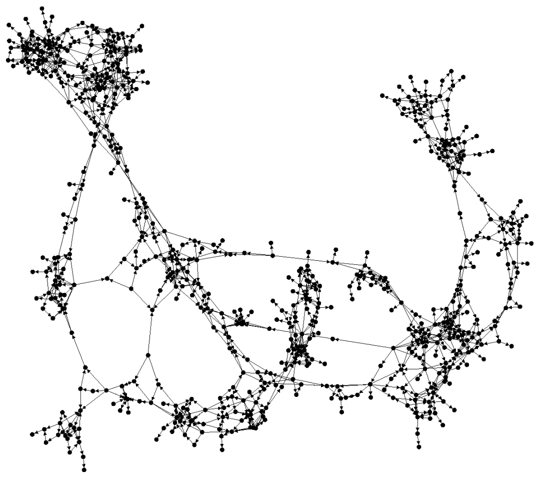
```
$n \leftarrow$ initialization number
$G(nodes, arcs) \leftarrow GraphGenerator(n)$
foreach $currentNode \in G.nodes$ do
    tag($currentNode$, “variable”)
    $enteringArcs \leftarrow \{arc \in G.arcs, isHeadOf(arc, currentNode)\}$
    if $enteringArcs \neq \emptyset$ then
        // currentNode is an output variable
        $modelNode \leftarrow$ new node
        tag($modelNode$, “model”)
        $G$.add($modelNode$)
        $G$.createArcBetween($modelNode$, $currentNode$)
        foreach $arc \in enteringArcs$ do
            $tailNode \leftarrow arc.tail$
            $G$.removeArc($arc$)
            $G$.createArcBetween($tailNode$, $modelNode$)
        end
    end

    // check for criteria
    $rand \leftarrow drawRandomNumber()$
    if $rand \leq criteriaProba$ then
        $critNode \leftarrow$ new node
        $G$.createArcBetween($currentNode$, $critNode$)
        if $rand \leq drawRandomNumber/2$ then
            tag($critNode$, “objective”)
        else
            tag($critNode$, “constraint”)
        end
    end
end
```

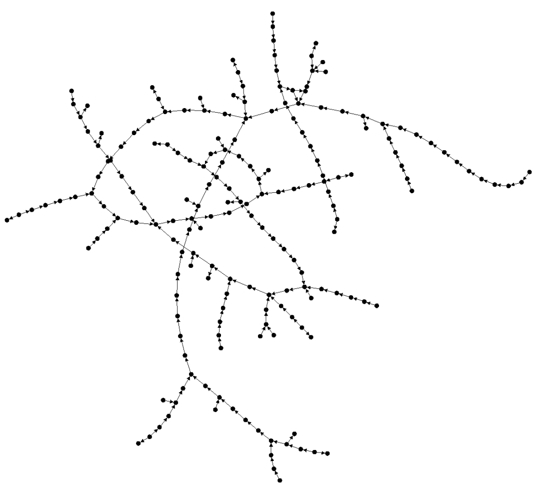
It must be noted that the final size of problems generated by this algorithm is not fixed, and can be significantly larger than the initialization value n. However problems produced using the same initialization number and the same graph generator are of comparable sizes.

We used graph generators proposed by the GraphStream library [1]. On Figure 12.1 some examples of graphs produced using these generators are visualized using GraphStream visualization tools. It should be noted that, because of the modifications we apply on the graph after it is created using the generator, the final agent graph does not necessarily respect the properties of the initial graph , however we can see on Figure 12.1 that the modifications we apply do not radically modify the characteristic topologies of the different graph types, which retain their characteristic topologies.

[1] http://www.graphstream-project.org/


(a) Random Euclidean graph example 1.


(b) Random Euclidean graph example 2.


(c) Small-World graph example 1.

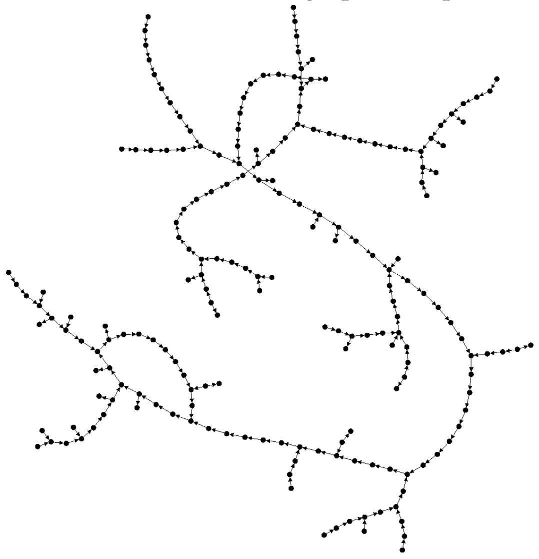
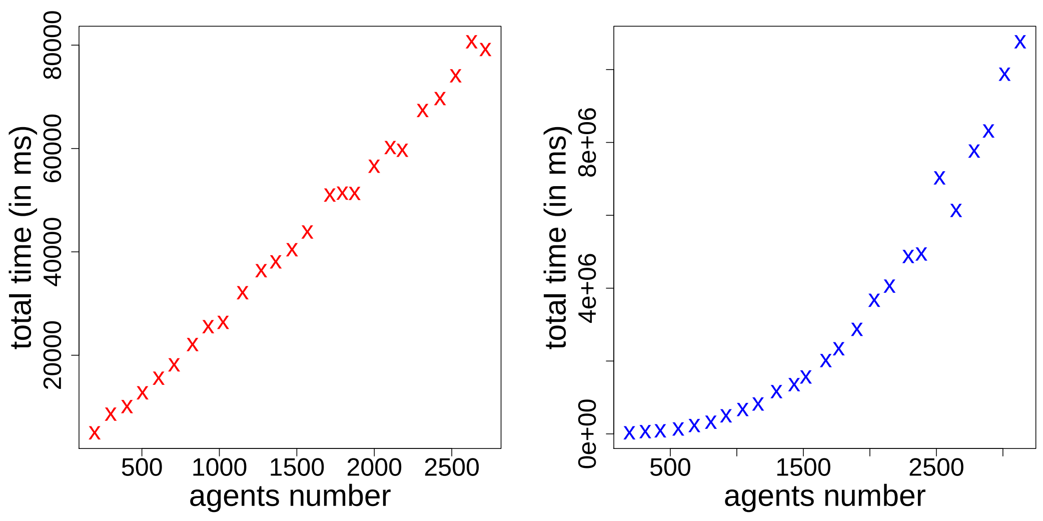

(d) Small-World graph example 2.

Figure 12.1: Examples of graph generation.

12.1.2 Experimental Results

Before discussing the results we obtained, let us add a word of warning concerning measuring real-time performances of Java applications (Java being the programming language used to implement our prototype). The Java Virtual Machine (JVM), which executes Java programs, applies a lot of complex optimization steps *during* the execution process. Consequently making some precise and non-biased measurements can be hazardous. Regarding our experiments, we tried to mitigate possible bias by taking the two following precautions:

- ▷ before running the measured experiments, running multiple problems whose results were discarded, in order to "heat" the JVM and allowing it to apply its optimization procedures beforehand.
- ▷ running the different experiments multiple times and in different orders, to "spread"



(a) Small-World graphs.

(b) Random Euclidean graphs.

Figure 12.2: Time performances by MAS size.

the benefits of possible optimizations happening at runtime.

We believe that these precautions are sufficient to obtain sufficiently meaningful results. Nevertheless, the reader should be warned not to consider the presented values as exact measurements of the system performances.

On Figure 12.2 are presented the results of the execution time of problems of different sizes. We generated agent graphs of different sizes using both Small-World and Random Euclidean algorithms. On the figure are presented the median time needed for all the agents of the problem to execute 800 behavior cycles. Interestingly, while the time performances in regard of small-world based problems graphs increase linearly with the size of the problems, the time needed in the case of random problem graphs seems to increase exponentially.

This seemingly poor performance can however easily be explained by looking at Figure 12.3, on which are shown the median degree (that is, the number of arcs entering of exiting a node) of the nodes in each generated problems. We can see, that, whatever the size of the problem, the median degree of a node in Small-World problems is mostly constant. However, in the case of Random Euclidean graphs, the median degree increases linearly with the size of the problem. Consequently, the exponential increase in time regarding Random Euclidean graphs can be explained by the conjugated effects of the increase in number of agents and the increase of the neighborhood size of the agents (whose impact in the behavior algorithm complexity has been exposed in section 6.3).

In order to corroborate this analysis, we studied the impact increasing the nodes degree has on the execution time of the MAS. On Figure 12.4, we show the results of another experiment using agent graphs generated using a Barabasi-Albert generator, which has the advantage of being able to easily generate graphs with different median degree sizes at the cost of only a slight increase of node numbers. On the figure is shown the average time for all the agents to make a behavior cycle in function of the median degree of the agents (using Barabasi-Albert based graphs of sizes between 510 and 550 nodes). We can see that the neighborhood size of the agents (the degree of the node) has the predicted effect on the

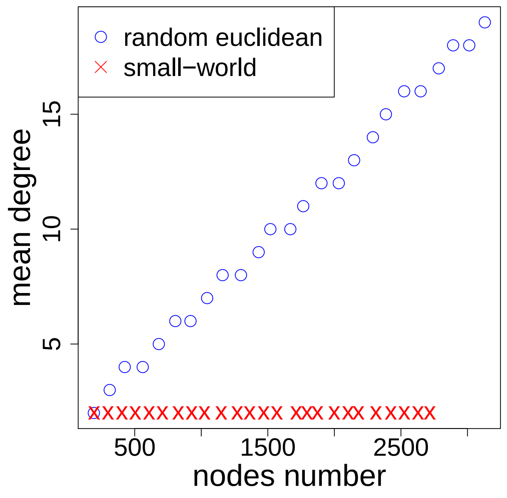
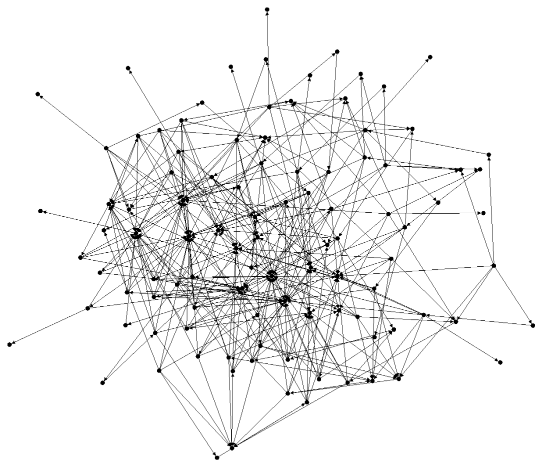
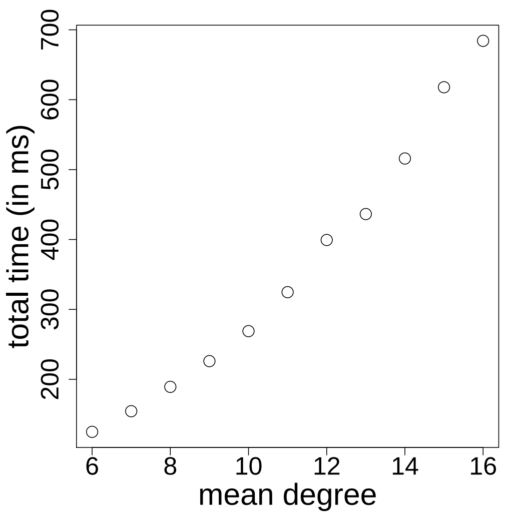


Figure 12.3: Comparison of nodes median degree by graph type.



(a) Barabasi-Albert graph example.



(b) Average time for a step (by median degree).

Figure 12.4: Time performances by node degree.

time needed by the agents to execute their behavior.

12.1.3 Analysis of Performances

Overall, we can conclude by observing these results that the time required for the agents to make a behavior cycle is not meaningfully impacted by the size of the problem. This property is an expected benefit of restricting the agents to local perception and decision process.
The results also shown the impact of increasing the neighborhood size of the agents. The time increase corresponds to the analysis we made concerning the computational complexity of the agent behavior (in 6.3). This analysis illustrates once more the importance of maintaining the behavior of the agents at a local level. Had the agent decision process involved not only

its immediate neighbors, but also a large group of agent, the computational cost would have increased even more sharply with the size of the problem.
As a remark, let us add that our implementation of the agent behavior was far from optimal from a performances point of view. In order to modify and experiment more easily on the agents, we voluntary kept separated into distinct modules the different solving mechanisms of the agents. A more efficient implementation could factor several of the treatments in order to reduce the computational complexity, obtaining a non-negligible performances improvement in the process.

12.2 Springs Networks

The automated agents graphs we detailed in the previous sections are useful to study raw scalability performances on different problem sizes. They are limited, however, by the fact that the problems they represent are meaningless. Consequently this kind of automated graphs are not useful to study the performances of the system in regard to the convergence toward a correct solution.

In this section, we propose a method to automatically generate another kind of problems graphs. The idea is to use graphs representing springs networks, where the edges linking the vertices represent springs whose extremities are tied together. Some of the nodes are fixed to arbitrary positions, while the goal of the problem is to find the position of the free nodes for which the spring network is stable, that is, where all the forces of the springs connected to a node are at an equilibrium.
The interest of such problem is that the springs network can be generated using classical graph generator algorithms, and that there is a unique (as long as at least one node has a fixed position) and easily verifiable solution.

12.2.1 Representing Springs Networks with NDMO

A major difference with the previous generated problems is that, in the case of the springs networks, the generated graph does not represent directly the agents graph. We must propose a NDMO modeling of the springs network in order to be able to apply our method.

The springs networks can be represented in 1, 2 or even more dimensions, each additional dimension adding a new coordinate for the position of the nodes. Interestingly, each dimension is independent in the sense that the strengths applied by the springs to the nodes can be computed independently[2]. Consequently the agents graph representing a spring network in n dimensions will be composed of n independent connected components.

We present here the procedure to generate the NDMO graph *for springs of one dimension*, which can easily be generalized to n dimensions:

- ▷ First of all the springs network is generated using a given graph generator.
- ▷ Each spring (*i.e.* the edges of the graph) is given a random force constant. In our experiments the forces were drawn randomly between 1 and 100.
- ▷ Each node has a random chance of becoming a fixed node (in our experiments this probability was fixed to 0.3). If the node is fixed, its current coordinate is set as a

[2]This is only true because we make the simplifying hypothesis that the springs can have a size of zero

constant value.

- For each free node, a variable agent is created, representing its coordinate.
- For each spring end linked to a free node, a model agent is created, representing the force applied to the node connected at this spring end, based on the spring constant and the coordinate. The exact computations for calculating the force of a spring on a linked node n_1 is $(x_2 - x_1) \times k$, where x_1, x_2 are the coordinates of the nodes linked to the spring and k is the initial force of the spring. This model takes in input the value of the variable agents of the linked nodes (if one of the node is fixed, its coordinate is used as a constant). For each model agent, an output agent is created representing the output value of the model.
- For each variable agent, a model agent is created, representing the sum of the spring strengths applied to the node. This model agent takes in input the output agents representing the strengths applied by the linked springs.
- For each of these model agents, an output agent is created. To this output agent is linked a constraint agent representing the constraint that the value of this output must be equal to 0 when the springs strengths are at equilibrium.

The generation procedure is synthesized in algorithm 12.2.

An example of transformation is presented on Figure 12.5. On Figure 12.5a is shown a simple springs network with a fixed node A and two free nodes B and C. The nodes are linked by edges 1 and 2. The corresponding agents graph (for one dimension) is shown on Figure 12.5b.
Notice how, since the node A is fixed, the model $force_{1,B}$ only takes the coordinate of B in input. It can also be seen that, since B is linked to two springs, the model sum_B, representing the sum of strengths applied to B, takes two values as inputs while, since C is linked to only one spring, sum_C only takes one input.

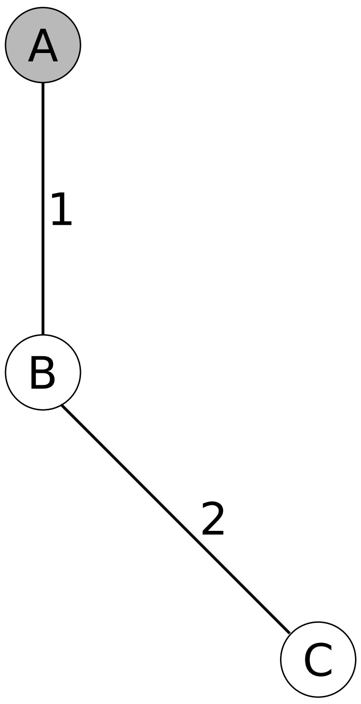
12.2.2 Springs Networks Experiments

In this section we study the impact of the problem size on the convergence speed of the system. To this end we created 2D springs networks generated with the previously detailed procedure, using Small-World graph generators to obtain graphs of different sizes. In each instance we executed the system until it found the correct solution (*i.e.* the positions of the nodes for which the system is at equilibrium). On Figure 12.6 is shown the result of our experiments. Each dot corresponds to a problem instance, with the agents number and the number of evaluations required to find the correct solution. We also show the linear regression trend line corresponding to the data set as a red dotted line.

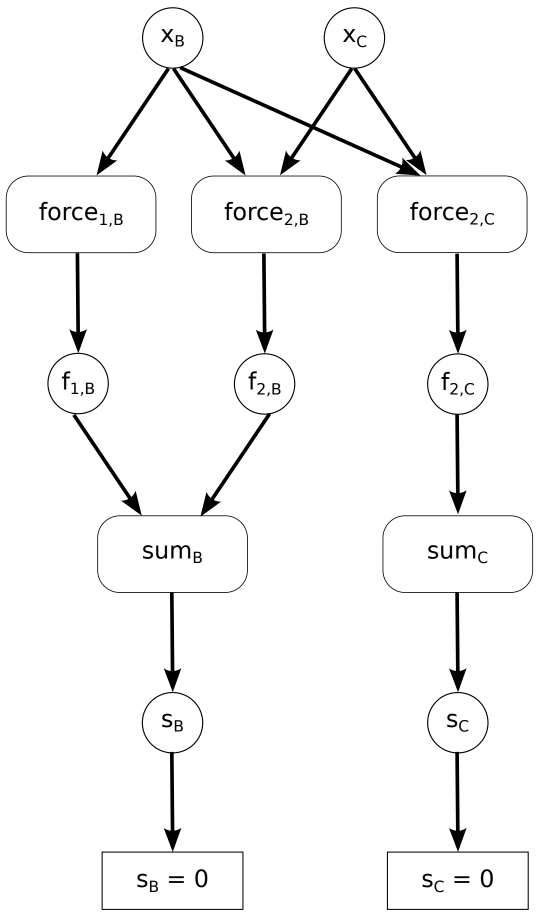
We can see that, while the number of evaluations can vary greatly for two instances of similar sizes, the number of agents has a relatively low impact on the convergence time of the system. These two observations are consistent with our previous analysis, confirming that the convergence is greatly impacted by the topology of the problem, but that the system is able to scale with the size of the problem.

Algorithm 12.2: Springs network problem generation

```
$n \leftarrow$ initialization number
$G(nodes, edges) \leftarrow GraphGenerator(n)$
foreach $e \in G.edges$ do
    $e.initForce \leftarrow random(1, 100)$
end
foreach $n_i \in G.nodes$ do
    $rand \leftarrow random(0, 1)$
    if $rand \leq 0.3$ then
        // fixed node
        tag($n_i$, “fixed”)
        // create constant $c_i$
        $c_i \leftarrow$ coordinate of $n_i$
    else
        // free node
        tag($n_i$, “free”)
        $x_i \leftarrow$ new variable agent
    end
end
foreach $e_i \in G.edges$ do
    foreach $n_i \in e_i.linked\_nodes$ do
        if $e_i.n_i$ is free then
            // create new force model for this free node
            $force_{i,n_i} \leftarrow$ new force model agent
            link($x_{n_i}$, $force_{i,n_i}$)
            if $e_i.n_{j \neq i}$ is free then
                // if the other node is free, add it as input..
                link($x_{n_j}$, $force_{i,n_j}$)
            else
                // ...else add it as constant
                addConstant($c_{n_j}$, $force_{i,n_i}$)
            end
            $f_{i,n_i} \leftarrow$ new output agent
            link($force_{i,n_i}$, $f_{i,n_i}$)
        end
    end
end
foreach $x_i \in$ Variable agents do
    // create sum of forces model agent
    $sum_i \leftarrow$ new sum model agent
    foreach $e_j \in n_i.edges$ do
        link($f_j$, $sum_i$)
    end
    $s_i \leftarrow$ new output agent
    link($sum_i$, $s_i$)
    // create constraint
    $c_i \leftarrow$ new constraint agent
    link($s_i$, $c_i$)
end
```



(a) Spring network.



(b) Corresponding agents graph (for one dimension).

Figure 12.5: Example of springs network transformation.

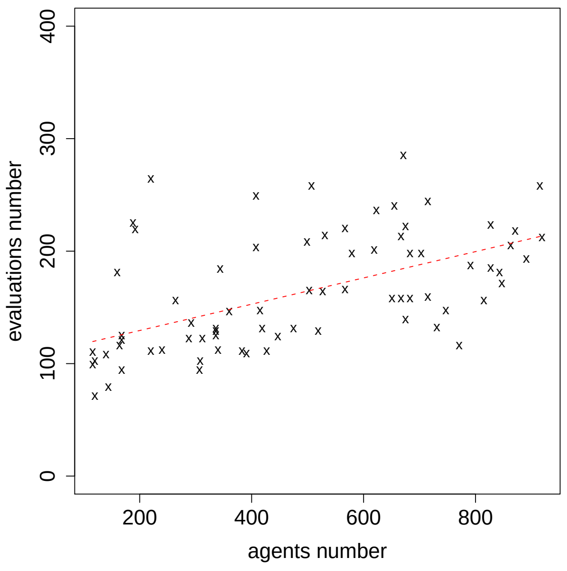


Figure 12.6: Number of evaluations required by agents number.

Analysis of Experiments and Future Works

In this part we presented several of the experiments we did on our system. These experiments show the validity of our agents behaviors on classical optimization problems, as well as the good performances of our system compared to other complex problems optimization methods. We also demonstrated the adaptation capabilities of our system, as well as its extensibility using uncertainties.
Using automatically generated problems, we were able to study the scalability properties of our system, and the impact of various criteria on its performances. The observations we made were coherent with our analysis of the agents behaviors.

Of course, the results presented in this part can only show a limited view of our system. For this reason, we continue our experiments in the different categories of test cases we have presented.
New academics test cases are experimented upon in order to detect potential new NCSs. In the context of the ID4CS project, we are currently in a validation phase using real-world large optimization problems, provided by our industrial partners (such as the one illustrated on Figure 12.7).
We also continue to work on new ways to experiments on optimization under uncertainties using different optimization schemes. Simultaneously, we continue our comparison work by applying our system on additional benchmark test cases.
At last, we have seen that the scalability properties of the system are influenced by the topology of the agents graph. For this reason we pursue our scalability experiments on generated test cases, using addition graph generation algorithms in order to study more in depth the effects of the different criteria.

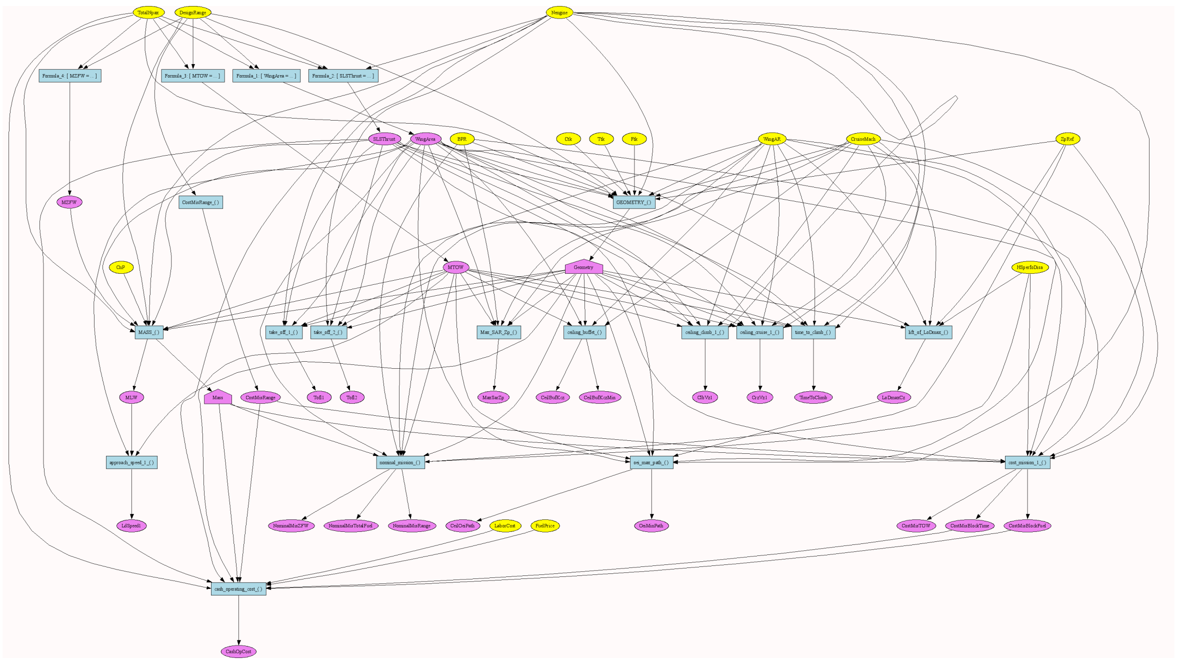


Figure 12.7: Example of ID4CS test case validation.

Conclusion and Perspectives

In this thesis we identified a severe limitation of current continuous optimization methods regarding the handling of complex continuation problem. Problems of this category are usually too complex to be solved by classical optimization methods because of multiple factors: the interdependencies of their components, their heavy computational cost, their nonlinearities *etc.* This limitation has been the motivation to propose new specific methods which divide the problem into several disciplines and distribute the optimization process using discipline-level optimizers. However these methods are often difficult to put in practice and cumbersome, not suiting the need of a flexible and iterative process often associated with such problems.

Thesis Contributions to Continuous Optimization

This thesis proposes a **new approach for solving complex continuous problems using an adaptive multi-agent system**. This system, designed following the Adaptive Multi-Agent Systems theory, proposes a decentralized way to automatically distribute the optimization process among the agents, and is able not only to solve large-scale complex problems, but also to adapt to changes made by the user during optimization.
The scalability of our approach is made possible by the fact that the agents keep a local view of the system. The system adapts to changes by propagating them from neighbor to neighbor, enabling the **interactive co-design of the solution**.

This system is built upon a **general continuous problem modeling we named Natural Domain Modeling for Optimization**, which transform the optimization problem into an entities graph. **This transformation does not require any simplification, modification or reformulation of the original problem and is fully automatic**.

Following the AMAS theory, we kept the agent perceptions and capabilities at a local level, allowing them to communicate and interact only with their immediate neighbors. Doing so, we are able to handle the problems complexity, as each agent keeps a local point-of-view. The agents are able to use external optimization tools (or can alternatively use local approximation techniques) to solve their local optimization problems. **This local optimization behavior, along with the message-based global consistency, enables a nominal distributed optimization process**.

We identified several configurations that are susceptible to disturb the good functioning of our system optimization process, corresponding to Non Cooperative Situations (NCS) of the AMAS theory. While these NCS can arise from the interaction of several agents, the agents had to be able to detect and solve them using only local mechanisms. **For each NCS we proposed local cooperative behaviors for the agents to be able able to cooperatively solve the situation and restore the correct optimization flow**. These cooperative behaviors use specific mechanisms and measures in order to correctly identify the NCS and take the adequate corrective action.

We proved the modularity of our design by showing how our system could be modified to handle additional concerns. To illustrate this, we **integrated in the agent mechanisms for managing the uncertainties propagation**, effectively allowing the system to realize optimization under uncertainties.

At last, we validated our approach on several test cases. We also integrated our system into a prototype in the context of the ID4CS project funded by the French National Research Agency, prototype currently tested by our industrial parters Airbus and Snecma.

Thesis Contribution to Multi-Agent Systems

We already introduced how our identification of specific NCS led to the development of specific agent mechanisms in order for the system to maintain a correct behavior, such as cooperative trajectories, cycle solving *etc.* Based on these NCS and solving mechanisms, we proposed more general Collective Problem Solving Patterns (CPSP). **These CPSP provide some guidance to the MAS designer regarding some potentially problematic agent configurations which may happen in the system, and propose some solving mechanisms to handle these situations.** We illustrated these CPSP with blueprints summarizing the configuration and mechanisms involved.
This work contributes to the ongoing effort of creating general tools for the design of AMAS in the context of problem solving, regrouped under the name AMAS4Opt (Adaptive Multi-Agents Systems for Optimization).

Using the Make Agents Yourself framework, we proposed **a modular agent architecture adapted to the modeling of hierarchical AMAS agent roles**, based on reusable “building blocks”. This architecture use a composition of stackable “skills” components, which allows for an efficient implementation of the handling of the different NCS by the agents.
This architecture can be used by the MAS designer as a base to design his own agents, using existing components, and contribute to the effort of providing tools for MAS engineering.

We proposed **a general graph representation of continuous optimization problems, which can be re-used by MAS designers as a common base to propose other MAS-based approaches for continuous optimization**. Using this common representation, such methods will be easier to compare in terms of solving mechanisms and performances.

Scientific Perspectives

Perspectives on MAS for Continuous Optimization

In regard of continuous optimization, our system could be enhanced with several additional capabilities. Currently, our system concentrates on providing *one* optimal solution. An obvious improvement would be to modify the agents to explore the Pareto front and provide several optimal solutions to the problem. A possible lead would be to modify the objective agents handling of criticality, in order to modulate the weight associated with each objective.

Another possible improvement for designers would be to integrate multi-fidelity models in the system. Multi-fidelity model is a technique to reduce the computational cost of optimization, using several versions of the same model with different computational costs. The low-cost, imprecise models are used at the start of the optimization process, while the high-cost, high-fidelity models are used when the system is starting to converge, in order to improve the precision of the solution. A difficulty of such mechanisms is the maintaining of the consistency between the different levels. This kind of mechanisms could be implemented in our system through the behavior of model agents.

The use of external optimizers could be improved by providing automated optimizer selection mechanisms, in order for the agents to be able to select the most appropriate optimizer, or even change of optimization method during the solving process. Such improvement would possibility require the creation of an optimization ontology in order to characterize the different optimizers.

One could imagine to add self-organizing capabilities to the system, in order to automatically compose the agent graph representing an optimization problem. Such functionality could be used, for example, to provide assistance to the designer during the specification of the optimization problem.
With such mechanisms, the user would only have to define a base of elements (variables, models, criteria *etc.*) and the system would be able to automatically assemble the elements into an agent graph, possibly creating (or requesting to the user) new elements as needed. Such functioning would be made possible more easily by the fact that the NDMO formalism does not necessitate any specific formulation to be valid.

Perspectives on the Design of MAS and CPSP

Concerning the design of MAS, we believe that both researchers and engineers would benefit greatly from the creation of an agent patterns repository. We intend to provide a more detailed and standardized description of the Collective Problem Solving Patterns we identified, with the goal of having a self-sufficient specification document.

An obvious continuation of this work is the identification of new CPSP, either concerning configurations we failed to identify in our application, or with configuration which appear in other application domains. Another possibility concerns the development of alternate handling mechanisms for existing CPSP.

An interesting addition to the CPSP would be to provide an implementation of specialized components which only need to be completed by providing specific functions based on the application domain. To which extend such partial instantiation of the CPSP is possible is an open question at this moment.

Perspectives on the AMAS Theory

At last, we would like to finish by discussing some interesting observations concerning the AMAS theory in itself and the identification of NCS. Prior works using the AMAS theory concentrated on the identification of NCS at a given moment. The detection of the NCS was usually done immediately by the agents, and the corrective actions were relatively simple and direct.

Most of the NCS we identified were quite different in their functioning. These NCS not only require the agents to cooperate over several iterations to be solved, but their identification itself requires the agents to take additional measures. Moreover, some of the configurations we identified are not systematically problematic in themselves, but only *potentially* problematic depending on some of the parameters (naturally converging cycles, hidden dependencies with adequate influences *etc.*). The common point among these situations is how they are related to the *dynamics* of the system, about its evolution toward one direction or another.

This observation leads to the question of whether a possible distinction could be made between *spatial* NCS, corresponding to the interactions between agents at a given instant, and *temporal* NCS, corresponding to the evolution of an agent over time. If such distinction proved to be relevant, it could lead to new insights on the design of AMAS-based systems, and possibly on the AMAS theory in itself.

An Adaptive Multi-Agent System for
Self-Organizing Continuous Optimization

Appendix

Author's Bibliography

International Conferences and Workshops with Referenced Proceedings

Tom Jorquera et al. "A Natural Formalism and a Multi-Agent Algorithm for Integrative Multidisciplinary Design Optimization". In: *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT), Atlanta, USA, 17/11/2013-20/11/2013*. 2013.

Tom Jorquera et al. "A Self-Adaptive Multi-Agent Algorithm for Interactive Continuous Optimization". In: *5th International Conference on Computational Collective Intelligence Technologies and Applications (ICCCI), Craiova, Roumanie, 11/09/2013-13/09/2013*. 2013.

International Conferences and Workshops without Referenced Proceedings

Tom Jorquera et al. "Relevance of an Adaptive Multi-Agent System Approach for Integrative Multidisciplinary Design Optimization". In: *European Workshop on Multi-Agent Systems (EUMAS 2012), Dublin, Ireland, 18/12/2012-19/12/2012*. 2012.

Tom Jorquera et al. "A Natural Formalism and a Multi-Agent Algorithm for Integrative Multidisciplinary Design Optimization". In: *International Workshop on Optimisation in Multi-Agent Systems at the Twelfth International Conference on Autonomous Agents and Multi-agent Systems (AAMAS), Saint Paul, Minnesota, USA, 06/05/2013-10/05/2013*. 2013.

Tom Jorquera et al. "Experimenting on a Novel Approach to MDO using an Adaptive Multi-Agent System". In: *10th World Congress on Structural and Multidisciplinary Optimization (WCSMO), Orlando, Florida, USA, 19/05/2013-24/05/2013*. 2013.

National Conferences and Workshops without Referenced Proceedings

Tom Jorquera, Jean-Pierre Georgé, and Christine Régis. "Self-Organizing Multi-Agent System For MDO". In: *12e Congrès de la Société Francaise de Recherche Opérationnelle et d'Aide à la Decision (ROADEF'11), Saint-Etienne, 02/03/2011-04/03/2011*. 2011.

Demos and Posters

Tom Jorquera, Jean-Pierre Georgé, and Christine Régis. "An Adaptive Multi-Agent System for Integrative Multidisciplinary Design Optimization (demo and 2-pages paper)". In: *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2012), Lyon - France, 10/09/2012-14/09/2012*. 2012.

Tom Jorquera et al. "An Adaptive Multi-Agent System for Multidisciplinary Integrative Design Optimization (Poster)". In: *International Conference on Agents and Artificial Intelligence (ICAART) 2013, Barcelona, Spain, 15/02/2013-18/02/2013*. 2013.

Bibliography

[AA91] American Institute of Aeronautics and Astronautics. *Current State of the Art on Multidisciplinary Design Optimization (MDO)*. General Publication Series. American Institute of Aeronautics & Astronautics, 1991. URL: `http://books.google.fr/books?id=rbgkAQAAIAAJ`.

[Ada+11] Emmanuel Adam et al. "Agents Tasks Reallocation for Collaborative Urban Supply Chain Management". In: *Holonic and Multi-Agent Systems for Manufacturing*. Ed. by Vladimír Mařík, Pavel Vrba, and Paulo Leitão. Vol. 6867. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 215–224. ISBN: 978-3-642-23180-3. DOI: `10.1007/978-3-642-23181-0_21`. URL: `http://dx.doi.org/10.1007/978-3-642-23181-0_21`.

[AL02] N.M. Alexandrov and R.M. Lewis. "Analytical and computational aspects of collaborative optimization for multidisciplinary design". In: *AIAA journal* 40.2 (2002), pp. 301–309.

[All+06] Janet K Allen et al. "Robust design for multiscale and multidisciplinary applications". In: *Journal of Mechanical Design* 128 (2006), p. 832.

[AT10] Anne Auger and Olivier Teytaud. "Continuous Lunches Are Free Plus the Design of Optimal Optimization Algorithms". English. In: *Algorithmica* 57 (1 2010), pp. 121–146. ISSN: 0178-4617. DOI: `10.1007/s00453-008-9244-5`. URL: `http://dx.doi.org/10.1007/s00453-008-9244-5`.

[Aud+00] C. Audet et al. "A branch and cut algorithm for nonconvex quadratically constrained quadratic programming". In: *Mathematical Programming* 87.1 (2000), pp. 131–152.

[BBG09] Noélie Bonjean, Carole Bernon, and Pierre Glize. "Engineering Development of Agents using the Cooperative Behaviour of their Components". In: *MAS&S@ MALLOW* 9 (2009), p. 107.

[Ber+03] Carole Bernon et al. "Adelfe: A methodology for adaptive multi-agent systems engineering". In: *Engineering Societies in the Agents World III*. Springer, 2003, pp. 156–169.

[Ber+05] Carole Bernon et al. "Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology ". In: *Agent-Oriented Methodologies*. Ed. by Brian Henderson-Sellers and Paolo Giorgini. Vol. ISBN1-59140-581-5. NY, USA: Idea Group Pub, 2005, pp. 172–202.

[BT95] Paul T Boggs and Jon W Tolle. "Sequential quadratic programming". In: *Acta numerica* 4.1 (1995), pp. 1–51.

[Car10] Peter Cariani. "On the Importance of Being Emergent. Extended Review of "Emergence and Embodiment: New Essays on Second-Order Systems Theory" edited by Bruce Clark and Mark B. N. Hanson.Duke University Press, Durham, 2009." In: *Constructivist Foundations* 5.2 (2010), pp. 86–91. URL: `http://www.univie.ac.at/constructivism/journal/5/2/086.cariani`.

[CC57] Abraham Charnes and William W Cooper. "Management models and industrial applications of linear programming". In: *Management Science* 4.1 (1957), pp. 38–91.

[CK03] D. Corne and J. Knowles. "Some multiobjective optimizers are better than others". In: *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*. Vol. 4. 2003, 2506 –2512 Vol.4. DOI: `10.1109/CEC.2003.1299403`.

[CK07] David W Corne and Joshua D Knowles. "Techniques for highly multiobjective optimisation: some nondominated points are better than others". In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM. 2007, pp. 773–780.

[CM09] Ian R. Chittick and Joaquim R. R. A. Martins. "An asymmetric suboptimization approach to aerostructural optimization". In: *Optimization and Engineering* 10 (1 2009), pp. 133–152. DOI: `10.1007/s11081-008-9046-2`.

[Cra+94] Evin J Cramer et al. "Problem formulation for multidisciplinary optimization". In: *SIAM Journal on Optimization* 4.4 (1994), pp. 754–776.

[Dan98] G. Dantzig. *Linear programming and extensions*. Princeton university press, 1998.

[DC05] Xiaoping Du and Wei Chen. "Collaborative reliability analysis under the framework of multidisciplinary systems design". In: *Optimization and Engineering* 6.1 (2005), pp. 63–84.

[DMSGK11] Giovanna Di Marzo Serugendo, Marie-Pierre Gleizes, and Anthony Karageorgos. "Self-organising Software: From Natural to Artificial Adaptation". In: (2011).

[DS83] J.E. Dennis and R.B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1983. ISBN: 9780898713640. URL: `http://books.google.fr/books?id=RtxcWd0eBD0C`.

[Eng00] Thomas M. English. "Practical Implications of New Results in Conservation of Optimizer Performance". English. In: *Parallel Problem Solving from Nature PPSN VI*. Ed. by Marc Schoenauer et al. Vol. 1917. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 69–78. ISBN: 978-3-540-41056-0. DOI: `10.1007/3-540-45356-3_7`. URL: `http://dx.doi.org/10.1007/3-540-45356-3_7`.

[GCG99] Marie-Pierre Gleizes, Valérie Camps, and Pierre Glize. "A theory of emergent computation based on cooperative self-organization for adaptive artificial systems". In: *Fourth European Congress of Systems Science*. 1999.

[Gli01] Pierre Glize. "L'adaptation des systèmes à fonctionnalité émergente par auto-organisation coopérative". In: *Hdr, Université Paul Sabatier, Toulouse III* (2001).

[Glo89] Fred Glover. "Tabu search—part I". In: *ORSA Journal on computing* 1.3 (1989), pp. 190–206.

[Gu+00] Xiaoyu Gu et al. "Worst case propagated uncertainty of multidisciplinary systems in robust design optimization". In: *Structural and Multidisciplinary Optimization* 20.3 (2000), pp. 190–213.

[Haf85] Raphael T Haftka. "Simultaneous analysis and design". In: *AIAA journal* 23.7 (1985), pp. 1099–1103.

[Hen08] James Hendler. "Avoiding Another AI Winter". In: *IEEE Intelligent Systems* 23.2 (2008), pp. 2–4. ISSN: 1541-1672. DOI: `http://doi.ieeecomputersociety.org/10.1109/MIS.2008.20`.

[HLW71] Yacov Haimes, Leon Lasdon, and Davis. Wismer. "On a Bicriterion Formulation of the Problems of Integrated System Identification and System Optimization". In: *Systems, Man and Cybernetics, IEEE Transactions on* SMC-1.3 (1971), pp. 296–297. ISSN: 0018-9472. DOI: `10.1109/TSMC.1971.4308298`.

[Hol92] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence*. MIT press, 1992.

[HW05] Raphael T. Haftka and Layne T. Watson. "Multidisciplinary Design Optimization with Quasiseparable Subsystems". English. In: *Optimization and Engineering* 6 (1 2005), pp. 9–20. ISSN: 1389-4420. DOI: `10.1023/B:OPTE.0000048534.58121.93`. URL: `http://dx.doi.org/10.1023/B%3AOPTE.0000048534.58121.93`.

[Jac88] Van Jacobson. "Congestion avoidance and control". In: *ACM SIGCOMM Computer Communication Review*. Vol. 18. 4. ACM. 1988, pp. 314–329.

[JJSRRS98] Sobieszczanski J., Agte J. S., and Jr R. R. Sandusky. *Bi-Level Integrated System Synthesis*. Tech. rep. 1998.

[Kad11] E. Kaddoum. "Optimization under Constraints of Distributed Complex Problems using Cooperative Self-Organization". PhD thesis. Université de Toulouse, Toulouse, France, 2011. URL: `ftp://ftp.irit.fr/IRIT/SMAC/DOCUMENTS/RAPPORTS/TheseElsyKaddoum_2011.pdf`.

[Kar84] N. Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. STOC '84. New York, NY, USA: ACM, 1984, pp. 302–311. ISBN: 0-89791-133-4. DOI: `10.1145/800057.808695`. URL: `http://doi.acm.org/10.1145/800057.808695`.

[KE95] James Kennedy and Russell Eberhart. "Particle swarm optimization". In: *Neural Networks, 1995. Proceedings., IEEE International Conference on*. Vol. 4. IEEE. 1995, pp. 1942–1948.

[KJV83] Scott Kirkpatrick, D. Gelatt Jr., and Mario P Vecchi. "Optimization by simmulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[KL95] Andrew Kusiak and Nick Larson. "Decomposition and representation methods in mechanical design". In: *Journal of Mechanical Design* 117 (1995), p. 17.

[Koc+99] Patrick N Koch et al. "Statistical approximations for multidisciplinary design optimization: the problem of size". In: *Journal of Aircraft* 36.1 (1999), pp. 275–286.

[Kro+94a] Ilan M. Kroo et al. "Multidisciplinary Optimization Methods for Aircraft Preliminary Design". In: *AIAA 5th Symposium on Multidisciplinary Analysis and Optimization* (1994). AIAA 1994-4325.

[Kro+94b] Ilan M. Kroo et al. "Multidisciplinary Optimization Methods for Aircraft Preliminary Design". In: *AIAA 5th Symposium on Multidisciplinary Analysis and Optimization* (1994). AIAA 1994-4325.

[Ks11] Zhang Ke-shi. "Concurrent Subspace Optimization for Aircraft System Design". In: *Aeronautics and Astronautics*. Ed. by Max Mulder. InTech, 2011.

[KT51] Harold W Kuhn and Albert W Tucker. "Nonlinear programming". In: *Proceedings of the second Berkeley symposium on mathematical statistics and probability*. Vol. 5. California. 1951.

[KX08] Vladik Kreinovich and Gang Xiang. "Fast Algorithms for Computing Statistics under Interval Uncertainty: An Overview". In: *Interval / Probabilistic Uncertainty and Non-Classical Logics*. Ed. by Van-Nam Huynh et al. Vol. 46. Advances in Soft Computing. Springer Berlin Heidelberg, 2008, pp. 19–31. ISBN: 978-3-540-77663-5. DOI: 10.1007/978-3-540-77664-2_3. URL: http://dx.doi.org/10.1007/978-3-540-77664-2_3.

[LA08] Mian Li and Shapour Azarm. "Multiobjective collaborative robust optimization with interval uncertainty and interdisciplinary uncertainty propagation". In: *Journal of Mechanical Design* 130 (2008), p. 081402.

[LCG11] S. Lemouzy, V. Camps, and P. Glize. "Principles and Properties of a MAS Learning Algorithm: A Comparison with Standard Learning Algorithms Applied to Implicit Feedback Assessment". In: *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*. Vol. 2. 2011. DOI: 10.1109/WI-IAT.2011.190.

[Lep+10] Julien Lepagnot et al. "A new multiagent algorithm for dynamic continuous optimization". In: *International Journal of Applied Metaheuristic Computing (IJAMC)* 1.1 (2010), pp. 16–38.

[Lev44] Kenneth Levenberg. "A method for the solution of certain non-linear problems in least squares". In: *Quarterly Journal of Applied Mathmatics* II.2 (1944), pp. 164–168.

[Liu+06] Huibin Liu et al. "Probabilistic analytical target cascading: a moment matching formulation for multilevel optimization under uncertainty". In: *Journal of Mechanical Design* 128 (2006), p. 991.

[MA04] R Timothy Marler and Jasbir S Arora. "Survey of multi-objective optimization methods for engineering". In: *Structural and multidisciplinary optimization* 26.6 (2004), pp. 369–395.

[MA10] R.Timothy Marler and JasbirS. Arora. "The weighted sum method for multi-objective optimization: new insights". English. In: *Structural and Multidisciplinary Optimization* 41.6 (2010), pp. 853–862. ISSN: 1615-147X. DOI: 10.1007/s00158-009-0460-7. URL: http://dx.doi.org/10.1007/s00158-009-0460-7.

[Mar63] Donald W. Marquardt. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". In: *SIAM Journal on Applied Mathematics* 11.2 (1963), pp. 431–441. URL: http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal\&id=SMJMAP000011000002000431000001\&idtype=cvips\&gifs=yes.

[Mas51] Frank J Jr Massey. "The Kolmogorov-Smirnov test for goodness of fit". In: *Journal of the American Statistical Association* 46.253 (1951), pp. 68–78. URL: http://www.jstor.org/stable/2280095.

[McC+06] John McCarthy et al. "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955". In: *AI Magazine* 27.4 (2006), p. 12.

[Mic+07] Maged Michael et al. "Scale-up x scale-out: A case study using nutch/lucene". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE. 2007, pp. 1–8.

[ML04] Roger Mailler and Victor Lesser. "Solving Distributed Constraint Optimization Problems Using Cooperative Mediation". In: *Proceedings of Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*. IEEE Computer Society, 2004, pp. 438–445. URL: http://mas.cs.umass.edu/paper/355.

[ML12] Joaquim R. R. A. Martins and Andrew B. Lambe. "Multidisciplinary design optimization: A Survey of architectures". In: *AIAA Journal* (2012). (In press).

[Mod+06] Pragnesh Jay Modi et al. "ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees". In: *ARTIFICIAL INTELLIGENCE* 161 (2006), pp. 149–180.

[Nai+92] Vijayan N Nair et al. "Taguchi's parameter design: a panel discussion". In: *Technometrics* 34.2 (1992), pp. 127–161.

[Nik+04] Efstratios Nikolaidis et al. "Comparison of Probability and Possibility for Design Against Catastrophic Failure Under Uncertainty". In: *Journal of Mechanical Design* 126.3 (2004), pp. 386–394. DOI: 10.1115/1.1701878. URL: http://link.aip.org/link/?JMD/126/386/1.

[NM65] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (1965), pp. 308–313. DOI: 10.1093/comjnl/7.4.308. eprint: http://comjnl.oxfordjournals.org/content/7/4/308.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/7/4/308.abstract.

[NMRM00] Alexandrov Natalia M. and Lewis Robert Michael. *Analytical and Computational Aspects of Collaborative Optimization*. Tech. rep. 2000.

[Noe12] Victor Noel. "Component-based Software Architectures and Multi-Agent Systems: Mutual and Complementary Contributions for Supporting Software Development". anglais. Thèse de doctorat. Toulouse, France: Université de Toulouse, 2012. URL: `http://www.irit.fr/publis/SMAC/DOCUMENTS/RAPPORTS/TheseVictorNoel-0712.pdf`.

[OJ96] Greg MP O'Hare and Nick Jennings. *Foundations of distributed artificial intelligence*. Vol. 9. Wiley. com, 1996.

[Osy84] Andrzej Osyczka. "Multicriterion optimization in engineering with FORTRAN programs." In: *JOHN WILEY & SONS, INC., 605 THIRD AVE., NEW YORK, NY 10158, USA, 1984, 200* (1984).

[PG04] Gauthier Picard and Marie-Pierre Gleizes. "The ADELFE Methodology ". In: *Methodologies and Software Engineering for Agent Systems*. Ed. by Federico Bergenti, Marie-Pierre Gleizes, and Franco Zambonelli. Klüwer Academic Press, 2004, pp. 157–176.

[PLB04] Ruben E Perez, Hugh HT Liu, and Kamran Behdinan. "Evaluation of multidisciplinary optimization approaches for aircraft conceptual design". In: *Proceedings of the 10th AIAA/ISSMO multidisciplinary analysis and optimization conference, AIAA paper*. Vol. 4537. 2004.

[PPS72] Vilfredo Pareto, . Page Alfred N., and Ann S. Schwier. *Manual of political economy / by Vilfredo Pareto ; translated by Ann S. Schwier and Alfred N. Page*. English. Macmillan London, 1972, xii,504p : ISBN: 0333135458.

[PW64] C. Van de Panne and A. Whinston. "Simplicial methods for quadratic programming". In: *Naval Research Logistics Quarterly* 11.3-4 (1964), pp. 273–302.

[Qui00] Joël Quinqueton. "Emergent Problem Solving in Multi-Agent Systems". In: *Zeszyty Naukowe* 10 (2000), pp. 147–163.

[RC04] Christian P Robert and George Casella. *Monte Carlo statistical methods*. Vol. 319. Citeseer, 2004.

[Roc+13] Robin Roche et al. "Multi-Agent Technology for Power System Control". In: *Power Electronics for Renewable and Distributed Energy Systems*. Ed. by Sudipta Chakraborty, Marcelo g. Sim
oes, and William e. Kramer. Springer, Apr. 2013. Chap. 15, pp. 567–609. ISBN: 978-1-4471-5103-6. DOI: `10.1007/978-1-4471-5104-3{\string_}15`. URL: `http://link.springer.com/chapter/10.1007/978-1-4471-5104-3_15#`.

[Ros60] H. H. Rosenbrock. "An Automatic Method for Finding the Greatest or Least Value of a Function". In: *The Computer Journal* 3.3 (1960), pp. 175–184. DOI: `10.1093/comjnl/3.3.175`. eprint: `http://comjnl.oxfordjournals.org/content/3/3/175.full.pdf+html`. URL: `http://comjnl.oxfordjournals.org/content/3/3/175.abstract`.

[Rou08] Sylvain Rougemaille. "Ingénierie des systèmes multi-agents adaptatifs dirigée par les modèles". français. Thèse de doctorat. Toulouse, France: Université Paul Sabatier, 2008. URL: `ftp://ftp.irit.fr/IRIT/SMAC/DOCUMENTS/RAPPORTS/TheseSRougemaille_1008.pdf`.

[Sac+89] Jerome Sacks et al. “Design and analysis of computer experiments”. In: *Statistical science* 4.4 (1989), pp. 409–423.

[SBR96] RS Sellar, SM Batill, and JE Renaud. “Response surface based, concurrent subspace optimization for multidisciplinary system design”. In: *AIAA paper* 714 (1996), p. 1996.

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. Wiley, 1998.

[SF02] K. Sentz and S. Ferson. *Combination of evidence in Dempster-Shafer theory*. Citeseer, 2002.

[Sha76] Glenn Shafer. *A mathematical theory of evidence*. Vol. 1. Princeton university press Princeton, 1976.

[SN08] Dieter Scholz and M Nita. “Preliminary sizing of large propeller driven aeroplanes”. In: *Proceedings of the RRD-PAE 2008 Conference, Brno, October 16-17*. DAR Corporation. 2008.

[SP05] Moon-Kyun Shin and Gyung-Jin Park. “Multidisciplinary design optimization based on independent subspaces”. In: *International Journal for Numerical Methods in Engineering* 64.5 (2005), pp. 599–617. ISSN: 1097-0207. DOI: `10.1002/nme.1380`. URL: `http://dx.doi.org/10.1002/nme.1380`.

[Sta88] Wolfram Stadler. *Multicriteria Optimization in Engineering and in the Sciences*. Vol. 37. Springer, 1988.

[Str+09] R. Stranders et al. “Decentralised coordination of continuously valued control parameters using the max-sum algorithm”. In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems. 2009, pp. 601–608.

[Tsu92] Kwok-Leung Tsui. “An overview of Taguchi method and newly developed statistical methods for robust design”. In: *Iie Transactions* 24.5 (1992), pp. 44–57.

[VFM96] R. Viennet, C. Fonteix, and I. Marc. “Multicriteria optimization using a genetic algorithm for determining a Pareto set”. In: *International Journal of Systems Science* 27.2 (1996), pp. 255–260.

[VRAC11] Meritxell Vinyals, Juan A. Rodríguez-Aguilar, and Jesús Cerquides. “A Survey on sensor Networks from a Multiagent Perspective”. In: *The Computer Journal* 54 (2011). In press. Published on-line on February 2010. DOI:10.1093/comjnl/bxq018., pp. 455–470.

[Wei99] G. Weiß. *Mutiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. Intelligent Robotics and Autonomous Agents Series. Mit Press, 1999. ISBN: 9780262731317. URL: `http://books.google.fr/books?id=JYcznFCN3xcC`.

[Wel+06] Jean-baptiste Welcomme et al. “Self-Regulating Multi-Agent System for Multi-Disciplinary Optimisation Process”. In: *European Workshop on Multi-Agent Systems (EUMAS), Lisbon*. Vol. 14. 12. 2006, pp. 2006–15.

[Wil03] Anthony Wilden. *System and structure: Essays in communication and exchange*. Psychology Press, 2003.

[WM05] D.H. Wolpert and W.G. Macready. "Coevolutionary free lunches". In: *Evolutionary Computation, IEEE Transactions on* 9.6 (2005), pp. 721 –735. ISSN: 1089-778X. DOI: 10.1109/TEVC.2005.856205.

[WM97] D.H. Wolpert and W.G. Macready. "No free lunch theorems for optimization". In: *Evolutionary Computation, IEEE Transactions on* 1.1 (1997), pp. 67 –82. ISSN: 1089-778X. DOI: 10.1109/4235.585893.

[Wol59] P. Wolfe. "The simplex method for quadratic programming". In: *Econometrica: Journal of the Econometric Society* (1959), pp. 382–398.

[WRB97] Brett A Wujek, John E Renaud, and Stephen M Batill. "A concurrent engineering approach for multidisciplinary design in a distributed computing environment". In: *Multidisciplinary Design Optimization: State-of-the-Art, Proceedings in Applied Mathematics*. Vol. 80. 1997, pp. 189–208.

[Yan12] Xin-She Yang. "Swarm-Based Metaheuristic Algorithms and No-Free-Lunch Theorems". In: *Theory and New Applications of Swarm Intelligence*. Ed. by Rafael Parpinelli and Heitor S. Lopes. 2012, pp. 1–16. DOI: 10.5772/30852.

[Yok+98] Makoto Yokoo et al. "The distributed constraint satisfaction problem: Formalization and algorithms". In: *Knowledge and Data Engineering, IEEE Transactions on* 10.5 (1998), pp. 673–685.

[Yok01] Makoto Yokoo. *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Ed. by O. Etzioni, T. Ishida, and N. Jennings. London, UK, UK: Springer-Verlag, 2001. ISBN: 3-540-67596-5.

[YSP08] S.I. Yi, J.K. Shin, and G.J. Park. "Comparison of MDO methods with mathematical examples". English. In: *Structural and Multidisciplinary Optimization* 35.5 (2008), pp. 391–402. ISSN: 1615-147X. DOI: 10.1007/s00158-007-0150-2. URL: http://dx.doi.org/10.1007/s00158-007-0150-2.

[Zad65] Lotfi Asker Zadeh. "Fuzzy sets". In: *Information and control* 8.3 (1965), pp. 338–353.

[Zad78] Lotfi Asker Zadeh. "Fuzzy sets as a basis for a theory of possibility". In: *Fuzzy sets and systems* 1.1 (1978), pp. 3–28.

List of Figures

1.1 Examples of local and global optima - *a* and *b* are global maxima, *c* is a local minimum, *d* is a local maximum and *e* is the global minimum. 8
1.2 Types of continuous optimization methods. 12
1.3 Convex and concave search spaces (from Oleg Alexandrov). 13
1.4 Illustration of the Nelder-Mead algorithm. 16

2.1 Illustration of the notion of Pareto Front (CC-BY-SA Johann Dréo). 28

3.1 Examples of aeronautics disciplines (source unknown, partially based on the work of C. W. Miller *Dream Airplanes*). 31
3.2 MDF method. 34
3.3 IDF method. 35
3.4 AAO method. 35
3.5 CSSO method. 36
3.6 CO method. 36
3.7 BLISS method. 37
3.8 ASO method. 38
3.9 MDOIS method. 38
3.10 QSD method. 39
3.11 Example of robust optimum (a) and non-robust optimum (b). 44

4.1 An example of the DCSP representation. 54
4.2 Illustration of functionally adequate and internal cooperative medium systems. 58
4.3 The conditions for cooperation and corresponding NCS. 59

5.1 Illustration of a Turbofan engine (CC SA-BY K. Aainsqatsi). 67
5.2 Class diagram of MDO problems. 68
5.3 Turbofan problem. 68
5.4 Model agent behavior. 72
5.5 Variable agent behavior: 1. receive request. 2. answer with inform. 73
5.6 Output agent behavior. 73
5.7 Constraint and objective agents behavior. 74
5.8 MAS class diagram. 74

6.1 Messages flow for simulation and solving. 78
6.2 Use of an external optimizer. 82
6.3 Conflicting trajectories example. 91
6.4 Criticality function of a constraint agent. 92
6.5 Criticality mechanism (the criticality of the requests is indicated in parenthesis). 93

6.6 Illustration of coordination requirements on a basin function with 3 different trajectories. 93
6.7 Cooperative trajectories for variable agents example. 96
6.8 Cooperative trajectories for model agent example. 98
6.9 Diverging cycle example. 100
6.10 Hidden dependency example. 102
6.11 Hidden dependency example (reproduced). 105

7.1 Uncertainties propagator as a black box function. 114

8.1 Overview of the ADELFE Method. 124
8.2 Actors and use cases identified during requirements studies for ID4CS. 125
8.3 AMAS agent architecture (as defined in ADELFE). 127
8.4 Example of MAY composition. 127
8.5 Agent Architecture — Behavior view. 128
8.6 *Skills* component internals. 130
8.7 Skills components dependencies tree with corresponding NCSs. 130
8.8 Agent Architecture — Communication view. 131
8.9 Agent Architecture — Monitoring view. 132
8.10 MAS Architecture. 133
8.11 User Interface of our prototype (problem canvas view). 135

9.1 class diagram of the Provider-Solicitor modeling. 139
9.2 Template on CPSP blueprints. 141
9.3 Conflicting Requests blueprint. 141
9.4 Cooperative Trajectories. 142
9.5 Cycle Solving blueprint. 143
9.6 Hidden Dependencies blueprint. 143
9.7 Asynchronous Requests blueprint. 144

10.1 Convergence of the Turbofan objectives for 100 random starting points. 152
10.2 Convergence of Viennet1 objectives for 100 random starting points. 152
10.3 Rosenbrock's valley. 153
10.4 Alexandrov problem. 154
10.5 Alexandrov agents behavior. 155
10.6 Convergence of the Alexandrov objective for 100 random starting points. 155
10.7 Sequential optimization trajectory. 158
10.8 Alexandrov agents behavior with perturbation (constraint change at dotted line).159
10.9 Turbofan agents behavior with perturbations (changes at dotted lines). 160

12.1 Examples of graph generation. 169
12.2 Time performances by MAS size. 170
12.3 Comparison of nodes median degree by graph type. 171
12.4 Time performances by node degree. 171
12.5 Example of springs network transformation. 175
12.6 Number of evaluations required by agents number. 175
12.7 Example of ID4CS test case validation. 176

List of Tables

6.1 AVT behavior 83
6.2 Conflicting Requests - Coordination Actions 95
6.3 Non Cooperative Situations Summary 107
6.4 Non Cooperative Situations Solving Mechanisms 108

10.1 Summary of experiments results for the tests cases 156
10.2 Uncertainties associated with the variables 157

11.1 Comparison Problem 1 – Simplicity 163
11.2 Comparison Problem 1 – Efficiency 163
11.3 Comparison Problem 1 – Accuracy 164
11.4 Comparison Problem 2 – Simplicity 165
11.5 Results on Comparison Problem 2 – Efficiency 165
11.6 Results on Comparison Problem 2 – Accuracy 166
11.7 Classification by criteria (based on [PLB04]) 166