ELYSER ESTRADA MARTÍNEZ

**A MULTI-AGENT SOFTWARE SYSTEM FOR REAL-TIME
OPTIMIZATION OF CHEMICAL PLANTS**

São Paulo
2018

ELYSER ESTRADA MARTÍNEZ

**A MULTI-AGENT SOFTWARE SYSTEM FOR REAL-TIME
OPTIMIZATION OF CHEMICAL PLANTS**

A thesis presented to the Polytechnic School
of University of São Paulo for the degree of
Doctor of Science

Advisor:

Prof. Dr. Galo A. Carrillo Le Roux

São Paulo
2018

ELYSER ESTRADA MARTÍNEZ

**A MULTI-AGENT SOFTWARE SYSTEM FOR REAL-TIME
OPTIMIZATION OF CHEMICAL PLANTS**

A thesis presented to the Polytechnic School
of University of São Paulo for the degree of
Doctor of Science

Advisor:

Prof. Dr. Galo A. Carrillo Le Roux

Concentration Area:

Chemical Engineering

São Paulo
2018

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

Martínez, Elyser Estrada

A MULTI-AGENT SOFTWARE SYSTEM FOR REAL-TIME
OPTIMIZATION OF CHEMICAL PLANTS / E. E. Martínez -- versão corr. -- São
Paulo, 2018.
168 p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo.
Departamento de Engenharia Química.

1.Real-Time Optimization 2.Multi-Agent Systems I.Universidade de São
Paulo. Escola Politécnica. Departamento de Engenharia Química II.t.

To my grandmother Blanca Nieves Cabrera Armas. She knows why....

ACKNOWLEDGMENTS

...and it was time to thank.

I want to thank first to professor Dr. Galo Antonio Carrillo Le Roux for the opportunity to study at the University of São Paulo (USP) and to count on his advisement. At the same time for his friendship, support and for trusting me since the very beginning.

It has been wonderful for me to return to the academy after ten years and to become student of the Chemical Engineering Department of the Polytechnic School. I'm a different person since I took the road through the offered courses. I thank to the university and the department staff. In that sense, I also render thanks to professor Dr. Ulises Javier Jáuregui Haza, from Cuba, for motivating me, been my first contact, and helping me to make my way.

To have the opportunity to be part of the multidisciplinary team that approached the project "Development of Real-Time Optimization solution with Equation Oriented approach", a research initiative of PETROBRAS, was an incredible experience. It opened my mind in an unimaginable way and put me in contact with a field I came to love and respect. I learned a lot from the presentations, interchanges with team members and watching them in action. I had also the chance to learn from experts of the industrial sector. Thanks to all of them.

A key piece of this work was built thanks to the interaction with professor Dr. Rafael de Pelegrini Soares, from the Chemical Engineering Department of the Federal University of Rio Grande do Sul (UFRGS). I really appreciate his help and contribution.

Family is an important component, present at every moment, even when distant. I would like to mention here the support I got from all my closest relatives. I'm specially grateful for the love and support from my lovely wife Arlen. She has been by my side in every second of this story. While still in the road, we were blessed with the born of Isabella, our wonderful daughter, that over anything made life more meaningful, pushing us to move on. I am grateful every single day for having her. A lot of thanks to my mom for all her love, concern and complicity, since ever. A particular acknowledgment also to Mine, for give us a hand in the final stage.

Finally, I need to say that I felt like at home in this country. That's why I want to conclude thanking to Brazil, for its warm welcome, all the opportunities and its incredible people. Special credits to my friend Jose Luis and his wife Cristina, for helping me and

my wife since we arrived to Brazil and supporting us in several aspects. In the same way a lot of thanks to our friends Alain, Giselle and Osmel, for their sincere friendship, motivation and enriching conversations at the USP.

To all them... I'm really thankful!

You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future...

Steve Jobs (Stanford Commencement Address, June 12, 2005)

RESUMO

Otimização em Tempo Real (OTR) é uma família de técnicas que buscam melhorar o desempenho dos processos químicos. Como esquema geral, o método reavalia freqüentemente as condições do processo e tenta ajustar algumas variáveis selecionadas, levando em consideração o estado da planta, restrições operacionais e os objetivos da otimização. Várias abordagens para OTR têm surgido da pesquisa acadêmica e das práticas industriais, ao mesmo tempo em que mais aplicações têm sido implementadas em plantas reais. As principais motivações para aplicar OTR são: a dinâmica dos mercados, a busca de qualidade nos resultados dos processos e a sustentabilidade ambiental. É por isso que o interesse em entender as fases e etapas envolvidas em uma aplicação OTR cresceu nos últimos anos. No entanto, o fato de que a maioria dos sistemas OTR em operação foram desenvolvidos por organizações comerciais dificulta o caminho para chegar nesse entendimento. Este trabalho analisa a natureza dos sistemas OTR desde o ponto de vista do software. Os requerimentos para um sistema genérico são levantados. Baseado nisso, é proposta uma arquitetura de software que pode ser adaptada para casos específicos. Os benefícios da arquitetura projetada foram listados. Ao mesmo tempo, o trabalho propõe uma nova abordagem para implementar essa arquitetura: Sistema Multi-Agentes (SMA). Dois protótipos de sistema OTR foram desenvolvidos. O primeiro aplicado num estudo de caso bem conhecido na literatura acadêmica. O segundo voltado para ser usado em uma unidade industrial. Os benefícios da abordagem SMA e da arquitetura, tanto na pesquisa relacionada com OTR, quanto na implementação em plantas reais, são analisados no texto. Um arcabouço de software que abrange os principais conceitos da ontologia OTR é proposto como resultado derivado do desenvolvimento. O arcabouço foi projetado para ser genérico, possibilitando seu uso no desenvolvimento de novas aplicações OTR e sua extensão a cenários muito específicos.

Palavras-chave: Otimização em Tempo Real (OTR). Arquitetura de Software (AS). Agente de software. Sistema Multi-Agentes (SMA).

ABSTRACT

Real-Time Optimization (RTO) is a family of techniques that pursue to improve the performance of chemical processes. As general scheme, the method reevaluates the process conditions in a frequent basis and tries to adjust some selected variables, taking into account the plant state, actual operational constraints and optimization objectives. Several RTO approaches have born from the academy research and industrial practices, at the same time that more applications have been implemented in real facilities. Between the main motivations to apply RTO are the dynamic of markets, the seek for quality in the process results and environmental sustainability. That is why the interest on deeply understand the phases and steps involved in an RTO application has increased in recent years. Nevertheless, the fact that most of the existing RTO systems have been developed by commercial organizations makes it difficult to meet that understanding. This work studies the nature of RTO systems from a software point of view. Software requirements for a generic system are identified. Based on that, a software architecture is proposed that could be adapted for specific cases. Benefits of the designed architecture are listed. At the same time, the work proposes a new approach to implement that architecture as a Multi-Agent System (MAS). Two RTO system prototypes were developed then, one for a well-know academic case study and the other oriented to be used in a real unit. The benefits of the MAS approach and the architecture, for researching on the RTO field and implementation on real plants, are analyzed in the text. A sub-product of the development, a software framework covering main concepts from the RTO ontology, is proposed as well. As the framework was designed to be generic, it can be used in new applications development and extended to very specific scenarios.

Keywords: Real-Time Optimization (RTO). Software architecture (SA). Software agent. Multi-Agent System (MAS).

LIST OF FIGURES

2.1	The plant decision hierarchy (taken from [12])	22
2.2	A typical steady-state MPA cycle	27
3.1	Pipes and Filters style (adapted from [104])	47
3.2	Blackboard style (adapted from [104])	48
3.3	Client-Server architectural style	50
3.4	Layered System architectural style	51
4.1	Starting point of the RTO system architecture design	59
4.2	RTO system architecture after applying Layered System style	60
4.3	RTO system architecture after PLANT DRIVER design	62
4.4	RTO system architecture after OPTIMIZER design	63
4.5	Architectural view of PROCESS INFORMATION SERVICE component	65
4.6	Architectural view of IMPROVER component	66
5.1	An agent in its environment (after [5])	68
5.2	Canonical view of an agent-based system (taken from [138])	70
6.1	Map of concepts covered by RTOF	84
6.2	RTO cycle implemented with the MAS.	95
6.3	Williams-Otto reactor and equations.	95
6.4	Environment created for the Williams-Otto RTO prototype	96
6.5	The RTO prototype running the Williams-Otto case study.	97
7.1	Schematic representation of the VRD process.	100
7.2	RTOMAS overall view	108
7.3	RTOMAS console. General system output.	120
7.4	RTOMAS console. OPTIMIZER agent output.	121
7.5	RTOMAS console. EMSOC artifacts in action.	121
7.6	RTOMAS console. ACTUATOR output	122
7.7	RTOMAS console. Optimization converged after 101 iterations.	123
7.8	RTOMAS web interface. Inspection of OPTIMIZER agent.	123
7.9	RTOMAS web interface. Inspection of ACTUATOR agent.	124
7.10	RTOMAS web interface. Inspection of TAGS BOARD artifact.	124

ACRONYMS AND ABBREVIATIONS

AI Artificial Intelligence.

ANN Artificial Neural Networks.

AOA Agent-Oriented Software Architecture.

AOP Agent-Oriented Programming.

AS Arquitetura de Software.

BDI Belief-Desire-Intention.

CAPE Computer Aided Process Engineering.

CARtAgO Common ARTifact infrastructure for AGents Open environments.

COIN Coordination, Organization, Institutions and Norms in Agent Systems.

CSTR Continuous-Stirred Tank Reactor.

DCS Distributed Control Systems.

DRPE Data Reconciliation and Parameter Estimation.

EML EMSO Modeling Library.

EMSO Environment for Modeling, Simulation and Optimization.

EO Equation oriented.

EVOP Evolutionary Operation.

FIPA Foundation for Intelligent Physical Agents.

GNM Gray-box Neural Models.

GUI Graphical User Interface.

ISOPE Integrated System Optimization and Parameter Estimation.

JaCaMo Jason + Cartago + Moise.

JADE Java Agent Development Framework.

JRE Java Runtime Environment.

LP Linear Programming.

MA Modifier Adaptation.

MAS Multi-Agent Systems.

MAS Multi-Agent System.

MOISE .

MPA Model Parameter Adaptation.

MPC Model Predictive Control.

OOP Object-Oriented Programming.

OTR Otimização em Tempo Real.

PDI Pentaho Data Integration.

PI Plant Information.

POP Pick Out Procedure.

QA Quality Attribute.

RTO Real-Time Optimization.

RTOF Real Time Optimization Framework.

RTOMAS RTO as a Multi-Agent System.

SA Software architecture.

SASO Self-Adaptive and Self-Organizing systems.

SCFO Sufficient Conditions for Feasibility and Optimality.

SDK Software Development Kit.

SLP Sequential Linear Programming.

SMA Sistema Multi-Agentes.

UML Unified Modeling Language.

USP University of São Paulo.

VRD Vapor Recompression Distillation.

TABLE OF CONTENTS

1	INTRODUCTION	17
1.1	Motivation	17
1.2	Proposition	18
1.3	Objectives	19
1.3.1	General Objective	19
1.3.2	Specific Objectives	19
1.4	Document organization	19
1.5	Congresses and Publications	20
2	REAL-TIME OPTIMIZATION	21
2.1	RTO in the Control Hierarchy	21
2.2	RTO Approaches	22
2.2.1	Centralized and Distributed RTO	22
2.2.2	Direct methods	24
2.2.3	Model-based methods	24
2.2.4	Classical RTO	26
2.2.5	ISOPE	27
2.2.6	Modifier Adaptation	28
2.2.7	Sufficient Conditions for Feasibility and Optimality	28
2.2.8	Other approaches	29
2.3	RTO Concerns	29
2.3.1	Process sampling	29
2.3.2	Steady-state detection	30
2.3.3	Data Treatment and Gross Error Detection	33
2.3.4	Data Reconciliation	34
2.3.5	Parameters Estimation	35
2.3.6	Optimization	37
2.3.7	Set-points update	38
2.4	Reported implementations and profitability	39
3	SOFTWARE ARCHITECTURE	40
3.1	Architectural Elements	41
3.1.1	Components	42
3.1.2	Connectors	42

3.1.3	Data	42
3.1.4	Configurations	43
3.2	Architecturally induced properties	43
3.3	System requirements and SA	44
3.4	Network based systems and peer-to-peer architectures	44
3.5	Architectural tactics, patterns and styles	45
3.6	Network-based architectural styles	47
3.6.1	Pipes and Filters	47
3.6.2	Replicated Repository	48
3.6.3	Shared Repository	48
3.6.4	Cache	49
3.6.5	Client-Server	49
3.6.6	Layered System and Layered-Client-Server	50
3.6.7	Code on demand	51
3.6.8	Remote Evaluation	52
3.6.9	Mobile Agent	52
3.6.10	Implicit Invocation	52
3.6.11	C2	53
3.6.12	Distributed Objects	53
3.6.13	Brokered Distributed Objects	54
3.7	Software Design Patterns	54
4	DESIGNING A RTO SYSTEM	56
4.1	RTO Application domain requirements	56
4.1.1	Functional requirements	56
4.1.2	Quality attribute requirements	57
4.1.3	Constraints	58
4.2	Architecture design process	58
4.2.1	First design iteration: starting from the Null Style	59
4.2.2	Second design iteration: a layered system	60
4.2.3	Third design iteration: first distributed objects	61
4.2.4	Fourth design iteration: deeping into PROCESS INFORMATION SERVICE	64
4.2.5	Fifth design iteration: deeping into IMPROVER	66
5	PREPARING THE WORKBENCH	67
5.1	Standars and Paradigms	67
5.1.1	Software Agents	67
5.1.2	Object Oriented Design and Programming	76

5.2	Technologies	77
5.2.1	EMSO	77
5.2.2	JAVA	78
5.2.3	JADE	79
5.2.4	JaCaMo	79
6	FIRST PROTOTYPE. APPLICATION TO A CASE STUDY	82
6.1	RTO ontology	82
6.2	Covering the RTO ontology	83
6.2.1	Framework description	84
6.2.2	Java Classes and Interfaces	85
6.3	A MAS in JADE for RTO	91
6.4	The MAS in action	94
6.5	Application to a case study	94
7	SECOND PROTOTYPE. APPLICATION TO A REAL CASE	99
7.1	Building the MAS	100
7.1.1	Implementing PLANT DRIVER component	101
7.1.2	Implementing OPTIMIZER component	103
7.1.3	RTOMAS overall view and dynamic	107
7.1.4	Implementing agents' mind	110
7.1.5	RTOMAS as a predefined organization of agents	119
7.1.6	Running RTOMAS	119
8	DISCUSSION	125
8.1	A software architecture	125
8.2	A software framework for RTO	126
8.3	A workbench for RTO research	127
8.3.1	Approaching global and distributed optimization	127
8.4	RTO with software agents	128
8.5	RTO approaches in a high level language	128
8.6	RTO as an open system	129
9	CONCLUSIONS	130
10	IDEAS FOR FUTURE WORK	131
	Appendix A Steady-state model for the Williams-Otto reactor	145
	Appendix B EMSO Flowsheet for the Williams-Otto case	148

Appendix C	EMSO Optimization for the Williams-Otto case	149
Appendix D	Steady-state model for the REPLAN case	150
Appendix E	EMSO Flowsheet for the REPLAN case	153
Appendix F	EMSO Parameters Estimation for the REPLAN case	155
Appendix G	EMSO Experiment Data File for the REPLAN case	156
Appendix H	EMSO Optimization for the REPLAN case	157
Appendix I	MPA OPTIMIZER agent source code	158
Appendix J	ACTUATOR agent source code	162

1 INTRODUCTION

Although pressures to operate a chemical plant as economically as possible have increased in recent times, it is not a new concern for engineers. For decades, the off-line steady-state optimization of chemical processes has been employed, in particular during the design of new plants. Nevertheless, once a plant is in operation, changes in raw material and energy costs, product values and equipment behavior in general means that the optimal operating conditions will change over time. An on-line optimization scheme is then necessary to keep process efficient [1].

On-line optimization is a term mentioned since the 70s. Advances in the speed and power of computers at lower costs have made it a more effective and attractive method for reducing processing costs. In [2], early methods for on-line optimization were described with practical application in industry. Nowadays, under the modern term of Real-Time Optimization (RTO), it is considered a Computer Aided Process Engineering (CAPE) method [3].

RTO seeks to optimize the plant operating conditions at each moment, improving the process performance. For that purpose, it reevaluates frequently the process conditions. Considering operational constraints, it tries to maximize the economic productivity by means of adjusting selected optimization variables, based on measurement data and in presence of disturbances and parameters uncertainty. That adaptation continues iteratively, trying always to drive the process as close as possible to the actual optimum.

1.1 Motivation

Historically, RTO systems have been built as custom and monolithic programs, as the result of internal innovation efforts in organizations, without a well-designed and documented software architecture. As recognized by Braunschweig and Gani [4], transforming multicomponent CAPE methods like RTO into useful tools requires appropriate software architecture and design. That is why since the first decade of the 2000s, companies applying RTO in their processes are tending to move away from developing programs themselves towards using vendor provided software.

At the same time, the fully understanding of every RTO step and the seek for improvements and new methods have been constant and rising aims of industry and academy.

Nowadays, although proven commercial products for RTO exist (e.g., Simsci ROMeo[®], Aspen Plus[®]) featuring options for parametrization, their black-box style difficult the understanding of how operations are carried out. In addition, the dynamic inclusion of new on-line optimization approaches or a multi-approach application is, in the most part of them, tough to implement or not a feasible option at all.

The concerns mentioned before, were key research targets of a collaboration project between the University of São Paulo (USP) and PETROBRAS. They served as motivation to design a software architecture specifically for RTO programs. As a proof of concept of the architecture's feasibility, a system prototype attending the designed structures should be built based on a real case. The practical use of the prototype in research and innovation activities, and the application of robust versions of it in real facilities were key motives. The prototype should serves as an open workbench, where process modeling tasks could be performed using equation-oriented models, to test and evaluate several RTO approaches, even concurrently.

1.2 Proposition

The kind of programs that perform RTO can not be embedded into the so called *functional/relational* software class. That class can be defined as *the one that includes programs that take input, perform some transformation over that input, produce output and halt afterwards*. The *functional/relational* view regards programs as functions $f : I \rightarrow O$ from some domain I of possible inputs to some range O of feasible outputs [5].

RTO programs can be better defined as *reactive systems*. Instances of that class of programs cannot adequately be described by the *functional/relational* view. Their main proceeding is to perform a long-term interaction with its environment (the plant in this case). Therefore, they should be described in term of their on-going behavior. Examples of reactive systems are: computer operating systems, process control systems and on-line banking systems. From the software development point of view, reactive systems are much harder to engineer than functional systems [6].

This dissertation proposes a software architecture for building RTO programs as reactive systems, based on functional and quality attributes requirements identified as inherent to an RTO application. Another proposition is the implementation of the architecture applying concepts, techniques and paradigms from a particular class of reactive systems: *Software Agents*. In this sense, the thesis proposes the development of RTO programs as Multi-Agent Systems (MAS), which facilitates the implementation of several concerns and brings additional benefits. Finally, a basic software framework that covers several

concepts from the RTO ontology is also proposed, which can be used in the development of future systems.

1.3 Objectives

1.3.1 General Objective

The main objective of this work is to design an internal organization for RTO programs as reactive systems. At the same time, to validate the proposition of instantiating that organization by means of a MAS approach, analyzing its feasibility and benefits of application in research and industry.

1.3.2 Specific Objectives

- To identify functional requirements for an RTO system according to main concerns of the field.
- To identify quality attributes that an RTO system should feature according to its nature.
- To design a domain-specific software architecture for RTO programs, as reactive systems, that meets the identified functional requirements and enables quality attributes.
- To model concepts of the RTO ontology as software entities.
- To implement a prototype of an RTO system based on the proposed architecture and following the MAS approach.
- To apply the implemented prototype to a real problem.
- To identify the properties and features that make the developed prototype valuable for RTO implementations.

1.4 Document organization

This thesis is structured in ten chapters including the actual one. The second chapter presents a literature review about RTO, covering its approaches and main concerns. Chapter 3 contains a review about the study field of Software Architecture. The design process of a software architecture for a generic RTO system is detailed in chapter 4. Taking that architecture as base, a set of selected tools and technologies is presented in chapter 5 in order to implement the system. The application of concepts and technologies

in the development of an RTO system prototype for a study case is presented in chapter 6. Chapter 7 describes a more comprehensive implementation of the architecture and its application to a real case. Several contributions of this work are presented in that chapter. Chapter 8 contains a discussion about the covered ideas and results. Finally, chapter 9 and 10 collect the conclusions and ideas for future work respectively.

1.5 Congresses and Publications

- Oral presentation of the work “Design and Implementation of a Real-Time Optimization Prototype for a Propylene Distillation Unit” during the European Symposium on Computer Aided Process Engineering (ESCAPE 24), June 15-18, 2014, Budapest, Hungary.
- Publication of the paper “Design and Implementation of a Real-Time Optimization Prototype for a Propylene Distillation Unit” in the Computer Aided Chemical Engineering magazine, Volume 33, 2014, pages 1321-1326.
- The work “An Agent-Oriented Software Framework for Real Time Optimization Solutions Implementation” was accepted for a poster presentation at the 2015 AIChE Annual Meeting, November 8-13, Salt Lake City, UT, United States.

2 REAL-TIME OPTIMIZATION

The RTO problem can be abstractly stated as: Given an operating plant with measurements y and a set of manipulable inputs m , determine values for m as a function of time, which will maximize some selected measure of plant's profitability, while meeting operational constraints [7].

In plants where external disturbances are both relatively slow and have a notable impact on the economic performance of the process, on-line optimization is an option to be considered [1]. The main gain from this alternative is called the on-line benefit and resides in the increase of plant's profit and the reduction of pollutant emissions [8].

In most industrial processes, the optimal operating point constantly moves. On one side, in response to the changing equipment efficiencies, capacities and configuration, and on the other hand, due to changes in the global market [8]. The latter includes demands for products, fluctuating costs of raw materials, products and utilities, increasing competition, tightening product requirements, pricing pressures and environmental issues. The period over which these various changes can occur ranges from minutes to months. These all factors have motivated a high demand for methods and tools in the chemical process industry that could provide a timely response to the changing factors and enhancing profitability by reducing the operating costs using limited resources [9][10]. As those changes make the operations far from optimum, RTO becomes important in such scenarios.

Two factors that have made feasible the application of RTO on a large scale are the availability of Distributed Control Systems (DCS) for data acquisition and process control and the application of multi-variable controllers. At the same time, the decrease in costs of computer hardware and software, the increase of pollution prevention and energy costs have stimulated producers to improve and optimize their processes [8].

2.1 RTO in the Control Hierarchy

The overall control system of a productive process is a complex mechanism that includes several tasks. In order to handle complexity a well-accepted approach is to regroup the tasks in levels, and sort these levels in a hierarchical structure following a logical functional and temporal order. The functional order has to do with ensuring process safety, profitability and product quality. The temporal order tries to handle the different

change frequency of variables. RTO is the first level in a typical control hierarchy where the economics of the plant is addressed explicitly (Fig. 2.1). Its main objective inside the hierarchy is to provide ideal economic targets for the Model Predictive Control (MPC) layer, which is expected to maintain the plant under control at its maximum economic performance [11].

Figure 2.1 – The plant decision hierarchy (taken from [12])

The transfer of the new set-points values from the RTO layer to the controller layer can be done with an open loop or closed loop implementation [13]. In processes where small changes in set-points can significantly impact the process performance it is sound to use an open loop implementation. In that scenario, the new operating conditions are not enforced directly to the controller, but rather an experienced operator will decide about their feasibility and if they represent the true plant behavior. According to the operator decision, the new conditions can be put into effect or discarded. Otherwise, using a closed loop implementation, the new conditions received from the RTO layer are immediately applied by the controller.

2.2 RTO Approaches

Although a variety of techniques have been proposed for RTO, some general properties allow to classify them. Foremost two big groups can be identified regarding the optimization scope: *centralized* and *distributed* RTO. At the same time, two other groups can be distinguished according to the use of process models: *direct* and *model-based* methods [14].

2.2.1 Centralized and Distributed RTO

According to [15] there is no agreement on which is the best approach, global or distributed. *Centralized* RTO is the most common approach found in the literature. In this approach a model of the entire plant, subject to constraints, is used to optimize the

process as a whole, based on an objective function.

In practice the centralized approach often cannot be applied due to the size and complexity of the optimization problem. That is where distributed RTO comes to help and the overall optimization is broken into several local optimizations which are orchestrated by a coordination model [10] updated iteratively. The intention of distributed optimization is to decompose the large-scale plant into subsystems in order to reduce complexity. Subsystems are not independent of each other so interconnections between them are defined. This approach is also referred as modular or hierarchical optimization.

The price of input and output stream of each sub-problem should be adequately determined when decomposing a global optimization, aiming to maximize the whole process profit. Decomposition techniques are studied, as a mature operations research subfield, in order to know which is the best way to decompose the whole optimization problem. The general idea behind that is to relax global constraints that connect two or more relatively independent parts of the optimization problem. Two techniques stand out as the more used: *Lagrangian Relaxation* and *Augmented Lagrangian* [16]. Alternatives exist for these classic techniques, addressing their main limitations, but their application depends on the optimization problem [17].

Darby and White [10] proposed an overall control systems structure to deal with the distributed RTO and enunciated some advantages of that approach:

- Distributed optimization can be performed in a higher frequency as the method only has to wait for steady-state in each subsystem rather than the whole plant.
- The subsystems can be modeled in a more accurately way in the local optimizers than in a global optimizer because local models will be less restricted than a global model. Besides different models can have distinct optimization levels.
- The incorporation of on-line adaptation or on-line parameter estimation are easier to implement with local optimizers.
- Local optimizers are less complex and easier to understand, hence easier to maintain. Furthermore, a local optimizer causing some problem can be taken off-line to fix it, while the rest of the optimizers continue to function.

Bailey et al. [18] pointed out two major difficulties with the distributed approach:

- The constraint information being passed between local optimizers to avoid conflicts is not as efficient as that in a global approach.

- Considering only parts of the process in the local optimizers can lead to get inconsistencies in the update of the parameters.

Authors of [19] gave a detailed method of how to perform distributed optimization with functional uniformity and common economics and operational objectives.

2.2.2 Direct methods

Direct methods or model-free approaches perform the real time optimization directly on the process without explicitly using any model. They use measurements taken directly from the plant, even though it could be expensive to do that task, and apply an optimization guided by the process performance objective function.

Direct methods are discussed in [19]. The most known technique is called Evolutionary Operation (EVOP), dating back to the 1950s [20]. EVOP is a statistical method of systematic experimentation where small changes are applied to the process variables during normal operation flow. The introduced changes are not so large that could lead to non-conforming products, but are significant enough to find the optimum process ranges. After each trial of set-point changes, the objective function is measured at steady-state and its sensitivity is used to readjust the set-points, that is why this technique is also called on-line hill-climbing gradient search.

EVOP requires a large numbers of set-point changes and on-line sensitivity measurements in order to observe the behavior of the objective function, especially in noisy processes. It also has to wait for steady-state after each set point change. In the consulted literature authors conclude that the method is too slow and simplistic for on-line optimization. Early attempts of on-line optimization used this method and criticized it for its slowness [15].

Direct methods are suggested when troubles arise in obtaining either first principle or empirical process models [14][8][21]. According to [22] direct methods present two major drawbacks, the first one related to the process dynamics and the second to noise and gross error presence.

2.2.3 Model-based methods

A model-based approach is essentially one where the optimization is performed on a model instead of the real system and the results are applied as set-points to the real system. At a glance, this approach is straightforward enough but it has the issue that any model-reality differences will generally result in sub-optimum process performance [23]. The used

model must represent changes in the manipulated process variables given the set-points calculated during optimization. The effects of these variables on the dependent ones are approximated by the model [15]. According to [7] model-based approaches have proven to be superior compared with direct methods, in spite of the problems associated with modeling inaccuracies.

When planning to use a model as base for an RTO implementation, two aspects have to be considered carefully. The first of them is about deciding whether to use a rigorous plant model or a simple one. The second has to do with deciding whether to build a phenomenological or a black-box model. The model does not necessarily need to be predictive, therefore it can be less expensive to develop.

To use a rigorous plant model for RTO has the disadvantage of requiring significantly longer computation time. On the other hand, to employ a too simple model can lead to an inaccurate representation of the plant behavior—i.e. a plant-model mismatch—and its optimization may result in non-optimal or infeasible operating conditions being calculated. This may arise from several causes: actual phenomena being modeled too simplistically or even not at all, uncertain model parameters whose real values might differ from those estimated or which change with time and process operating conditions [1].

Phenomenological or first-principle models representing the physical and chemical laws that describe the major events in the plant are preferred over a general black-box model. This is because their wider range of validity and bigger set of physically meaningful variables to identify. However, a good compromise should be made between the simplicity of the model and the range of validity [24].

First-principle models were first used in support of process control applications in the 1960–1970s, coinciding with the initial use of digital computers supporting process operations. Some of these models were set up in order to generate gain information for steady-state optimizers based on Linear Programming (LP) or Sequential Linear Programming (SLP). However, several issues were faced at that time with the use of phenomenological models due to the lack of powerful modeling tools and difficulties to program computers. That is why black-box models were used in most of the cases that suffered of significant numerical noise. In spite of that, these initial model-based applications delivered value [25].

Equation oriented (EO) methodologies for building first-principle models started to appear in the 1980s. EO approaches allowed performing optimization separating the development of complex models from the solution method. Due to the availability of partial derivative

information and the use of solution methods with super linear or quadratic convergence rates they offer better computational efficiency. A large number of on-line phenomenological steady-state optimization applications were implemented in the petrochemical and refining industry in the 1990s [25].

Model's quality has a large impact on the RTO scheme success. The concept of model adequacy for RTO was introduced by [26][23], giving criteria for choosing an adequate model. They defined a point-wise model adequacy criterion as the model's ability to have an optimum that coincides with the true plant optimum. A procedure with analytical methods for checking the point-wise model adequacy was developed in that work. The methods use reduced space optimization theory, and from them, more practical numerical methods were developed for industrial applications [15][27].

Typically, a plant operates at steady-state most part of the time, with transient periods that are relatively short compared to steady-state operations. Therefore, depending on the frequency of the input disturbances and the time required by the process to settle down, a dynamic model may not be necessary and a steady-state model can be used to describe the plant. In virtually all practical cases, the steady-state model is non-linear [10].

2.2.4 Classical RTO

The availability of EO modeling environments, large scale sparse matrix solvers and the increase of computers processing capabilities allowed the implementation of the classical way to build RTO schemes in the late 1980s [12]. This strategy, also called Model Parameter Adaptation (MPA), uses a first-principle steady-state model to describe the process behavior and as constraints of the optimization of an economic objective function.

The basic idea behind MPA is to update some key parameters of the model to reduce the plant-model mismatch, using plant measurements, and then to optimize using the re-parameterized model [28]. The RTO cycle starts with the steady-state detection phase. Once a stationary point is identified, it goes through the data reconciliation and gross error detection stages. The resulting information is then used in the parameter estimation module to update the model's parameters. Finally, the updated model is employed to calculate the optimal operational conditions. The calculated values of those variables considered as set-points are sent to the process control layer aiming to maximize the plant profit. Described steps and the flow that links them are pictured using an Unified Modeling Language (UML) [29] activity diagram in figure 2.2.

Figure 2.2 – A typical steady-state MPA cycle

The absence of structural plant-model mismatch is not guaranteed with the use of high-fidelity plant model. Incomplete plant information and measurements noise are important sources of uncertainty in the updated parameters, increasing the plant-model mismatch [27]. Under large structural plant-model mismatch and small excitation in the operation conditions, the MPA method cannot guarantee convergence to the true plant optimum [30]. In spite of those vulnerabilities and numerical optimization issues, steady-state MPA is the most used on-line optimization approach by industry [12].

2.2.5 ISOPE

Aiming to handle the structural plant-model mismatch Roberts [31] proposed a modification of the classical RTO method called Integrated System Optimization and Parameter Estimation (ISOPE). This methodology integrates the parameter estimation and the optimization steps. ISOPE optimizes a modified economic function, adding a modifier term coming from the parameter estimation step that allows a first-order correction [27]. The idea is to complement the measurements used in the MPA method with plant derivative information whenever it can be calculated accurately. An important property of this technique is that it achieves the real process optimum operating point in spite of inevitable errors in the mathematical model employed in computations [22].

The main challenge in ISOPE is the requirement of plant derivatives used to compute the modifiers values, since the estimation of these quantities is considerably affected by the measurement noise [32]. Complexity increases geometrically with the problem dimension. Some methods have been proposed for estimating these process derivatives, between them: finite difference approximation [31], dual control optimization [33] and Broydens method and dynamic model identification method, with a linear [34] and nonlinear [35]

model.

An analysis of ISOPE is presented in [36]. Several versions of the ISOPE algorithm have been developed. A review of them can be found in [22].

2.2.6 Modifier Adaptation

Another technique to tackle the plant-model mismatch problem named Modifier Adaptation (MA) was developed by Marchetti et al. [30]. The MA approach adjusts the optimization problem by adapting linear modifier terms in the cost and constraint functions. These modifiers are based on the differences between the measured and predicted values of the constraints and cost gradients, i.e., quantities that are involved in the necessary conditions of optimality.

MA differs from the classical RTO method in the way plant information is used. The idea is to employ the measurements to fulfill the necessary first-order optimality conditions of the plant without updating the model's parameters. The cost and constraint predictions between successive RTO iterations are corrected in such a way that the point satisfying the Karush–Kuhn–Tucker [37] conditions for the model coincides with the plant optimum [30].

How the modifiers are calculated and the parameters updating is the fundamental difference between MA and ISOPE. MA calculates modifiers from the derivatives of economic objective function respect to inputs. The ISOPE method uses the derivatives of output respect to inputs; also, parameters are updated during ISOPE iterations while MA uses a fixed parameters set during optimization. The main limitation for industrial applications of MA is that the scheme needs an accurate plant gradient available in order to calculate the real plant optimum in the presence of plant-model mismatch [27].

2.2.7 Sufficient Conditions for Feasibility and Optimality

The methodology called Sufficient Conditions for Feasibility and Optimality (SCFO), was proposed in Bunin et al. [38][39]. The method adapts the nonlinear optimization theory to RTO problems. Based on plant derivative information and topology it tries to calculate the plant optimum executing a projection problem without violating any “hard” constraint. Given a possible optimal operational point, that could be predicted by any other RTO strategy, the SCFO method implements a correction in this target. A combination of the concepts of descent half-space and quadratic upper bound is employed to derive sufficient conditions, guarantying the improvement of the objective function value. Concepts of approximately active constraints and Lipschitz continuity are also combined

and used to ensure the constraint feasibilities at each iteration [27].

Assumptions like the knowledge of global Lipschitz constants, global quadratic upper bounds and the exact value of restrictions at current iteration, are very difficult to meet in practical applications [27]. The lack of accurate real process derivatives is also an issue in practice. With that in mind the method was modified. The modification consisted in the use of a feasible region for the plant gradient given by the derivative of the real process to guarantee a descent region. This way the algorithm works within a region where the worst case ensures a decrease in the plant objective function without violating the constraints. Even so, authors state that it is not clear if the application of SCFO is beneficial, since the algorithm may affect the convergence speed, especially when the RTO target is good.

2.2.8 Other approaches

An approach using Gray-box Neural Models (GNM) for RTO is presented in [40]. The models are based on a suitable combination of fundamental conservation laws and neural networks. Authors use the models in at least two different ways: to complement available phenomenological knowledge with empirical information, or to reduce dimensionality of complex rigorous physical models. A combination of genetic and nonlinear programming algorithms is used to obtain the optimum solution. Results showed that the use of GNM models, mixed genetic and nonlinear programming optimization algorithms is a promissory approach for solving dynamic RTO problems.

2.3 RTO Concerns

Several concerns are linked to RTO software systems implementation. Some of them are present in every scenario and others are specific to the used methodology. Next sections address those concerns present in most common situations or where the classical RTO approach will be implemented.

2.3.1 Process sampling

A RTO system can be considered as a data dependent one, as it depends on data to do its work. Therefore, how to get data from the real process is an important concern with three edges: which variables to sample, how frequently to sample, and how to get data from the plant supervisory and control structures.

Data is passed into the RTO system for different purposes. The most embracing of them could be resumed as to get a snapshot of the actual plant operational state, in order to

see if a better state could be found. That idea encompasses other more specific purposes that are directly related to the concerns described in the following sections. The choice of which process variables will be sampled, and at which frequency, is justified by those concerns and the RTO approaches and steps they are related to.

The last mentioned edge of process sampling embraces the concerns that could be found when trying to read real-time data from plant interfaces. The first decision system operators will deal with is about sampling raw process data, directly from sensing equipments, or data that has been treated by some system like Plant Information (PI) [41]. Sampling data from sensing devices directly, or without any treatment, has the property of bringing data noises that are native to a sensing action along with the actual data. That could be good for some RTO tasks where having real data variance is vital and bad for others. Same thing for getting smoothed data from plant systems. This edge also covers the technical aspects of interfacing with those systems or devices.

2.3.2 Steady-state detection

To identify when the process is close enough to steady-state is an important task for satisfactory control of many processes. If process signals were noiseless steady-state detection would be trivial: at steady-state there are no data value changes. However, process signals usually contain noise. Therefore, statistical techniques are applied to declare probable steady-state situations. Instead of considering just the most recent variable samples, steady-state detection methods need to observe the trend to make any assertion [42]. An automated or on-line approach for process status identification is preferred to human interpretation. At the same time, the method should be easy to understand by operators to troubleshoot the process [43].

Process status identification can be defined as a classification problem. Hence, type-I and type-II errors can appear in each classification iteration. Type-I error will be produced if the implemented method claim that the process is at transient-state when it is actually at steady-state. Claiming the process is at steady-state when it is actually in a transient state will be a Type-II error. Type-I errors can lead to false input to data reconciliation. Type-II errors or false detection can lead to misinterpretation of true process features, especially if the incorrect steady-state data are subsequently reconciled [44]. Fifteen issues associated with automatic steady-state detection in noisy processes were identified and described by Rinnehart [42]. According to the author, any practicable method needs to address all issues.

The literature collects a variety of techniques for on-line steady-state detection. A straight-

forward approach is to perform a linear regression over a data window and then perform a T-test on the regression slope. A slope significantly different from zero suggests that the process is not at steady-state [45]. This is normally an off-line technique. As the entire data window needs to be updated and the linear regression re-performed in each iteration, an on-line implementation has computational issues regarding data storage, computational effort and user expertise. With a long data window, the computational effort increases and detection of changes would be delayed. To use a short window can lead to a wrong analysis because of the noise, and the effective window length would change with variations in noise amplitude. A false reading could be obtained in the middle of an oscillation [46].

A geometric approach for the description of process trends was presented in [47]. A technique based on the calculation and comparison of data variances was proposed in [42]. A weighted moving average was used in that work to filter the sample mean. The filtered mean square deviation from the new mean is compared then with the filtered difference of successive data. To estimate the mean value a low pass filter is used. This method reduces significantly the computational requirements and is less sensitive to the presence of abnormal measurements. Moreover, using a weighted average to filter the calculated variances creates a delay in the characterization of process measurement frequency. These delays can be the cause of detection problems in periods where the signal properties vary in real time [44].

Wavelet transform features were used in [48] to approximate process measurements by a polynomial of limited degree and to identify process trends. The modulus of the first and second order wavelet transforms, which ranges between 0 and 1, were used in [49] as base for the detection of near steady-state periods. Steady-state is claimed when the statistic allocated in the value is nearly zero. This method can accurately analyze high frequency components and abnormalities.

A hybrid methodology based in the combination of Wavelet and statistical techniques is proposed in [44]. The authors use a method proposed in [50] and [49] as base for the wavelet features. The statistical component of the methodology is based on low pass filter and hypothesis test. The methodology shows to be efficient detecting pseudo steady-state operating conditions, reducing type-I and II errors in on-line applications.

Cao and Rhinehart [46] presented a method, which uses an F-like test applied to the ratio of two different estimates of the system noise variance. The estimates are calculated using an exponential moving-average filter. Data is filtered using a moving-average filter. For each of the filters a parameter is chosen between 0 and 1. Values of these parameters

are set based on relevance of actual values in comparison to the past ones, and could be interpreted as forgetting factors and express something analog to a window size [51]. The authors proposed the parameters to be tuned empirically and present some guidelines for that. If the ratio is close to one, then data can be considered in steady state. The method proved its computational efficiency and robustness to process noise distribution and non-noise patterns.

A procedure to assess the trend of a time series called Reverse Arrangements Test is described in [52]. The book also provides tables containing confidence intervals. A too big or too small value of the calculated statistic, compared to the standard values, could indicate a trend in data and the process should not be considered in steady-state. The test is applied sequentially to data windows.

Authors of [53] developed an algorithm for a filter to treat noisy processes data. LeRoux proposed an original usage of this filter [51]. Choosing a data windows containing an odd number n of elements the measurements series is filtered. Data is then interpolated using a polynomial of degree p , with $p < n$, obtaining this way less noisy information. Subsequently the first derivative of each polynomial is calculated at the central points and the value is used as a statistic for assessing the stationary character of the point. Steady-state is indicated when the statistic value is nearly zero. The signal is not scaled in this method by the noise level.

A method splitting a data window in half is presented in [54]. It calculates the mean and variance in each half and finds the ratio of the differences in averages, scaled by the standard deviations. When the ratio is equal to unity the process is considered to be at steady-state.

Authors of [55] and [56] cite the use of tests like the T-test, Wald-Wolfowitz runs, Mann-Whitney [57] and Mann-Kendall [58] [59]. Based on the T-test of difference between means of sequential data windows, the steady-state is claimed when the statistic value is nearly zero. The issue here is that the method can claim steady-state if an oscillation is centered between the windows. This method scales the difference in level by the noise standard deviation [42].

The method used by [60] consists in calculation of data variance over a moving window. If variance is large, the steady-state hypothesis is rejected. This method does not scale the signal by the noise level.

The Mahalanobis distance [61] is used as the statistic to declare the steady-state in [62].

If the measured distance is large, the steady-state hypothesis is rejected. Authors of [63] proposed a polynomial approach to identify trends in multi-variable processes. The process will be at steady-state if the measured trend error in the recent data window is small.

The standard deviation of a recent data window was calculated and compared to a threshold value from a nominal steady-state period in [64]. Case the standard deviation exceeds excessively the threshold, process is considered not at steady-state. It was noticed by the authors that determination of the essential variables, the window length and the threshold standard deviation for each variable are critical for the method success. A similar method is used in [65] comparing the standard deviation in a moving window to a standard value. Another technique described by Svensson in [65] is to calculate the residual from steady-state mass and energy balances, declaring not at steady-state when the residual is large.

Steady-state detection plays a crucial role in the implementation of RTO schemes that wait for an steady-state to run optimization. On those approaches, process data is passed onto the next RTO steps just once the plant is declared to be at steady-state. Parameter adjustment of models and data reconciliation should only be performed with nearly steady-state data, otherwise in-process inventory changes will lead to optimization errors [46].

2.3.3 Data Treatment and Gross Error Detection

Measured process variables are subject to two types of errors: random errors (which are generally assumed to be independently and normally distributed with zero mean) and gross errors caused by nonrandom events. Power supply fluctuation, as well as wiring and process noise are common sources of random errors. Instrument biases or miscalibration, malfunctioning measuring devices and process leaks are sources of gross errors in data [66][67][68]. All those factors lead to plant data not being used to their full potential. Therefore, measurements need to be cleaned of high frequency noise and abnormalities.

In [44], authors describe a denoising procedure based on Wavelet transformation. Using the temporally redundant information of measurements, random errors are reduced and denoised trends are extracted. These trends are considered more accurate than raw measurements.

The number of gross errors present in measured data is normally smaller than the amount of random errors. However, gross errors invalidate the statistical basis of reconciliation due to their non-normality. A series of small adjustments to other measured variables will be caused by one gross error present in a constrained least squares reconciliation. That is why gross errors need to be identified and removed before data reconciliation is

accomplished [15].

Presence detection of any gross errors, so that suitable corrective actions can be taken, is known as the gross error detection problem [69]. Several techniques have been proposed to approach this problem, based on the assumption that the measurements are a random sample of true values at steady-state. The most of them are based on statistical hypothesis testing of measured data [15]. Tests are generally based on linear or linearized models [67]. At least two alternative ways of estimating the value of variables—e.g. measured and reconciled values— are needed for gross error detection methods to work [66].

Statistic tests based on the residuals or imbalances of the constraints either individually (normal distribution test) or collectively (chi-square test) were proposed since some decades ago by Reilly and Carpani [70]. Wavelet features are used in [71] to identify and remove random and gross errors. In [72] and [73] a technique, which has been more recently named as the Maximum Power test for gross errors, is detailed. This test has greater probability of detecting the presence of a gross error, without increasing the probability of type-I error, than any other test would on a linear combination of measurements.

The Generalized Likelihood Ratio is another technique detailed in [74] and [66]. This method provides a framework for identifying any type of gross errors that can be mathematically modeled. This can be very useful for identifying other process losses such as a leak [15].

Details about other typical gross error detection algorithms can be found in [69], allocated in the [75], [76], [77] and [78]. Working out the tests for gross error detection, the probability of a correct detection must be balanced against the probability of mis-predictions.

2.3.4 Data Reconciliation

Both measured raw data and the resulted from a denoised procedure are generally inconsistent with process model constraints. Besides, due to inconvenience, technical non-feasibility or cost, not all the needed variables are generally measured. The adjustment of measured variables, minimizing the error in the least squares sense, and the estimation of unmeasured variables whenever possible so that they satisfy the balance constraints, is known as the data reconciliation problem.

Data reconciliation is only applied when the steady-state data sets are identified and extracted. Several approaches have been reported in literature for data reconciliation, some using just a mass balance and others based on a complete plant model [79].

Problems of data reconciliation and gross error detection, together, are sometimes called as data rectification [80]. Some works have addressed this problem as a whole. In one of the techniques, bounds are incorporated into data reconciliation [81]. As imposed inequality constraints can cause the data reconciliation solution no longer be obtained analytically, along with difficulties to perform gross error detection, an iterative cycle of data reconciliation and gross error detection were developed by the authors with good results.

A simultaneous data reconciliation and gross detection method was presented in [67]. The method consists in the construction of a new distribution function by means of the minimization of an objective function that is built using maximum likelihood principles. Contributions from random and gross errors are taken into account. This method proved to be in particular effective for non-linear problems [15].

Artificial Neural Networks (ANN) have also been used for data rectification. An example can be found in [80]. The work describes an ANN that is trained to perform data rectification based on a given model and constraints.

2.3.5 Parameters Estimation

Parameters estimation is an important concern that arises in model-based RTO schemes. Parameters are unknown quantities, usually considered constants but with the capacity to be variables, that are estimated starting from an initial knowledge and using data measurements [82].

Parameter estimation is the step after data reconciliation in which the reconciled values of the process variables are used to set values for the model parameters, pursuing to decrease the plant-model mismatch [83] [84]. The updated model is then used in the optimization step. Inside the RTO scheme, parameters estimation is seen as an on-line activity, involving the continual adjustment of the model, to take account of plant changes taking place during its operation. These modifications can occur due to process internal factors like heat transfer coefficients deterioration, equipment efficiencies degradation or, more generally, changes in various biases relating to temperature, pressure, flow or energy. All these changes can be considered unmeasured disturbances, which are normally observable with process data, have well understood effects on model sensitivity, and are expected to vary over time [25].

Two important issues emerge when developing an on-line parameter estimation scheme. Firstly, there is the need to know which of the uncertain parameters are the major con-

tributors to the plant-model mismatch and consequently need to be estimated. Usually, a sensitivity-based approach [85] is used to determine model parameters that have little or no effect on model predictions, and can therefore be either discarded or held at a fixed value for the purpose of estimation. The second issue has to do with which of the available plant measurements make up the best set to be used to estimate the parameters values [1].

Regarding how many measurements to use in order to estimate model's parameters there are differences among industrial practitioners. To build a square problem with equal number of measurements and parameters or to limit the number of measurements are choices of some of them. Others choose a larger number of measurements, selecting those that, via the model, interact with other measurements [12].

One trivial on-line parameters estimation approach is to compute additive or multiplicative biases after the model solves to match the process outputs. Simple additive offsets will not properly translate information in the measurements that indicate changes to process sensitivities. The use of multiplicative biases can be suitable in some cases but will change model sensitivity between the inputs and the biased outputs [25].

Another approach pairs a measurement with an individual disturbance variable establishing a dynamic relationship between these pairs, with a tuning factor to control the speed of matching the current process measurement. This can be interpreted as a nonlinear generalization of the feedback strategy used in linear MPC technologies. According to [25] this approach is simple to understand, and allows updating both input/output disturbance variables, but can be difficult to tune and lacks disturbance variable bounding capability, since it is not optimization based. Besides, it focuses on current measurement and does not consider the history of measured values for the on-line parameters estimation problem.

An improvement to this approach is the formulation of a state/disturbance estimation problem [86]. Authors of [25] mention approaches to state estimation those based on Extended Kalman Filtering [87] and its variants as predominant, and Moving Horizon Estimation as well. A comparison of these two techniques is presented in [88].

To perform data reconciliation and parameters estimations separately, as a two-step approach, is considered inefficient [89] and not statistically rigorous [90]. That has led to the development of simultaneous strategies for Data Reconciliation and Parameter Estimation (DRPE) [91]. The more generally used procedure for DRPE is the minimization of the least squares error in measurements, based on the assumption that measurements have normally distributed random errors in which case least squares is the maximum likelihood estimator [89]. The problem shows up when gross errors or biases are present in the data,

as these can lead to incorrect estimations and severely bias reconciliation of the other measurements.

Estimators derived from robust statistics can be used as objective functions in simultaneous DRPE problems. These estimators put less weight on large residuals corresponding to outliers, resulting in less biased parameter estimates and reconciled values [89]. Robust likelihood functions for the DRPE problem are presented in [92]. Commonly used robust estimators include the m-estimators [93] [94] [95] [96].

2.3.6 Optimization

The central goal of an RTO iteration consists in the calculation of the optimal operational point for the plant according to actual conditions. That task is accomplished in the step where an objective function is optimized using the plant model as restrictions. An optimization is also performed to solve the data reconciliation and parameter estimation problems. Usually a nonlinear programming solver is used.

Optimization methods can be divided into two categories: *Direct Search* and *Gradient-Based* methods [11]. *Direct Search* or *Global Optimization* are relatively simple compared to Gradient-Based methods. They are used in complex chemical processes where difficulties appear to build an accurate model, when calculation of objective function is quite hard or if the gradient of the objective does not exist or is computationally expensive.

The two main characteristics of direct search methods are: the gradient of the objective function is not approximated and only function values are used [97]. They can be further classified in exact and heuristic methods. Exact methods include *Interval Halving*, *Multi-Start* and *Branch and Bound* methods. Its effectiveness has been proved in finding the global optimum if they are allowed to run until the termination criteria are met [98].

Heuristic methods are based in rules, frequently derived from natural processes, to perform local and global searching trying to improve the current solution. According to [98] these methods are quite effective in calculating the global optimum. Genetic Algorithm, Simulated Annealing and the Tabu and Pattern Search are inside this group.

Gradient based methods need, at each iteration, some knowledge of the objective function gradient to find an improved direction that could lead to the optimal solution. Because of their tendency to converge to a local optimum in the presence of multiple function optima, these methods are also called local methods. That is why they depend on the choice of starting guesses and are inclined to stay in a local minimum if started far away from the

global optimum [99]. Gradient-based methods can be divided into two big groups: Analytical and Numerical methods. Analytical methods are based on the accurate calculation of the objective function gradient. At the solution, they satisfy the first and second order optimality conditions [99] [100].

Numerical methods use linear or quadratic approximations of the objective function and constraints. They convert the problem into a simple linear or quadratic programming problem. When it is not possible to calculate analytically the objective function gradient or constraints, a numerical approximation of the gradient is obtained by using finite difference type methods [99].

For finding local optima in large-scale problems, gradient-based methods have proven to be very fast and effective. Instances are: *Partition* method, *Lagrange Multiplier* method, *Successive Linear Programming*, *Sequential Quadratic Programming* and *Generalized Reduced Gradient*.

2.3.7 Set-points update

Once a candidate optimal operational point has been determined during optimization, values for those variables that turn out into set-points, need to be passed to the control layer. This is the last link of the RTO workflow chain and three main concerns can be identified here.

The first of them has to do with the way the set-points passing occurs in practice, determining the open-loop or closed-loop character of the RTO implementation. In scenarios where there is a high trustfulness in the RTO system and its suggestions, passing set-point values to the control layer can be an automatic task (closed-loop). In the other hand, if an analysis by a human expert needs to be done on RTO propositions before application, or the system is been used just for testing purposes, an open-loop strategy can be brought into play to decide if set-points will be updated or not.

Another concern has to do with the set-point alterations frequency. Applying new values every time a new set is obtained from the optimization phase will not always result in improving plant profit. According to [101], as the optimization phase is always affected by measurements noise and measured or unmeasured plant disturbances, an on-line statistical results analysis [28] will be required to decrease the frequency of unnecessary set-points alterations, increasing this way plant profits.

The third concern is about interfacing with the plant control layer and it has two dimensions:

the logical structure of the control layer and its technical implementation. Regarding logical structure, in some cases, just a regulatory control is present. In more sophisticated environments, an advanced control layer takes care of passing command actions to the regulatory layer. Depending on how hierarchical the control structure is, RTO would need to deal with more or less subjects, in its eagerness to lead the plant to a better operational condition. The technical dimension is about the diversity of technologies that can be found in a real facility. An appropriate implementation should be built in order to deal with legacy control systems interface.

2.4 Reported implementations and profitability

Although systems for on-line optimization have been implemented over a considerable period, their failure or success has several times gone unnoticed. As of 1983, authors of [9] estimated that the economic value added by the process could be improved by 3%-5% using on-line optimization techniques. In [79], examples are exposed of RTO implementations in large chemical plants in the period from 1983 to 1991, achieving profitability in the order of 1%-10%. Other industrial applications of on-line optimization were reported from 1992 to 1996 in refineries and chemical plants with a 5%-20% increase in profit.

The large amount of material being processed provokes that even from small improvements in the process performance the resulting economic payoffs are significant. Intangible profit in terms of a better understanding of plant behavior is also recognized. In [12] was reported that, as of 2011, there were more than 300 applications of RTO all over the world, spread in a variety of chemical and petrochemical processes. As of 2015, PETROBRAS reported 7 RTO implementations, 3 in distillation processes, 3 in cracking processes and 1 in utilities processes.

3 SOFTWARE ARCHITECTURE

Software architecture (SA) is a form of software design that occurs earliest in a system's development and at a very high abstraction level. The key issue of SA is organization. Other issues SA tackles are global control structure, protocols for communication, synchronization, and data access; likewise assignment of functionality to design elements and physical distribution. SA is essential to taming the complexity of a system. The architecture is a bridge between the goals of stakeholders and the final resulting system. Every software system has an architecture.

At the heart of SA is the principle of abstraction. The architecture is an abstraction of a system that selects certain details about it and suppresses others that are not useful for reasoning about it. A complex system will contain many levels of abstractions, each with its own architecture. That recursion of architectures continues down to the most basic system elements, those that cannot be decomposed into less abstract ones.

SA has been key target for software engineering research since the 1990s. The academy investigates methods for determining how best to partition a system, how components identify and communicate each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations [102].

In spite of the interest in SA as a field of research there is little agreement among researchers and organizations about what exactly to include in the concept of SA. The IEEE 1471 standard defines SA as “*the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution*”. Every system can be shown to comprise elements and relations among them to support some type of reasoning [103].

From a structuralist vision, Bass et al. [103] define SA as *the set of structures needed to reason about a computing system, comprising software elements, relations among them, and properties of both*. A structure is simply a set of elements held together by a relation. Systems can and do comprise more than one structure, each of them providing a different perspective and design handle. No single structure holds claim to being declared as *the* architecture and not all of them are architectural. According to [103] a structure is architectural if it supports reasoning about the system and its properties. That

reasoning should concern an attribute of the system that is important to some stakeholder.

Garlan and Shaw [104] describe an architecture of a system as a collection of computational components together with a description of the interactions between these components—the connectors. This model is expanded upon in Shaw et al. [105]: *The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system.* Further elaboration of this definition can be found in [106].

A modern definition says that an architecture is *an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.* In addition to levels of architecture, a software system will often have multiple operational phases, such as start-up, initialization, normal processing, re-initialization, and shutdown. Each operational phase has its own architecture [102].

A definition that uses the term *architectural element* as its kernel enunciates that SA is defined by a *configuration of architectural elements —classified as components, connectors, and data —constrained in their relationships in order to achieve a desired set of architectural properties.*

As can be verified in the above definitions, a recurrent term in them is *architectural elements*, referencing the main building blocks of an architecture. Next section tackles that subject.

3.1 Architectural Elements

The term *architectural element* encompasses all the buildings blocks used to create those structures. A generally accepted ontology of architectural elements group them into three classes: *components*, *connectors* and *data* [102]. The structure of the architectural relationships among them, during a period of system run-time, is defined by a *configuration*. Behavior of each architectural element is part of the architecture insofar as that behavior can be used to reason about the system.

3.1.1 Components

A component is *an abstract unit of software instructions and internal state, a unit of computation, that provides a transformation of data elements via its interface* [102]. Garlan & Shaw [104] define them as the system's elements that perform computation. Examples of transformations performed by component elements are calculations, translation to a different format, load into memory or storage from a secondary storage, generation of brand new data from several sources, etc.

Among other elements, components are the most easily recognized aspects of an architecture. Rather than been defined by its implementation, a component is defined by its interface and the services it provides to other components in the architecture.

Complexity of modern software systems have necessitated a greater emphasis on multicomponent systems, where the implementation is partitioned into independent components that communicate to perform a desired task.

3.1.2 Connectors

Connectors are used to model and govern interactions among components. A *connector* is *an abstract mechanism that mediates communication, coordination, or cooperation* among them [107]. They can be seen as the glue that holds together the architectural pieces [108].

Connectors enable communication between components by transferring data elements from one interface to another, without changing that data. They can have its own internal architecture designed to fulfill its mission. In the process, data can be transformed internally, transferred, and then have its transformation reversed before delivering. Therefore, from an external perspective, *connectors* do not transform data, in contrast with a *component's* behavior.

Some examples of *connectors* are: shared repositories and representations, remote procedure calls, message-passing protocols and data streams.

3.1.3 Data

A *datum* is an element of information that is transferred from a component, or received by a component, via a *connector*. *Data* elements carry semantic from one component to other, enabling the behavior of the system as a whole. Some examples include: messages, byte-sequences and serialized objects. *Components* can generate *data* by themselves.

3.1.4 Configurations

A *configuration* is a set of constraints for the architectural relationships among *components*, *connectors* and *data*, during a period of system run-time [102]. It can be seen as a connected graph that describes architectural structures and specifies restrictions on interactions between elements of those structures.

The main concept behind *configuration* is precisely the one of *constrain*. *Constraints* are frequently motivated by the application of a software engineering principle to an aspect of the architecture [109]. It is the *constraints* established by a *configuration* what induce a set of properties within an architecture.

3.2 Architecturally induced properties

Properties induced in a system by *configurations* can be classified in *functional* and *non-functional* properties. They are derived from the selection and arrangement of *components*, *connectors*, *data* and restrictions applied on their relationship.

Functionality is the ability of the system to do the work for which it was intended. It is the basic statement of the system's capabilities, services and behavior, perceived by stakeholders, users or consumed from another software. *Functionality* is the answer to the question: *Which are the system's responsibilities?*. The international standard ISO 25010 defines functional suitability as “*the capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions*”.

Functional attributes have the strangest relationship to SA, in the sense that they do not determine the architecture. Given a set of required functionalities, there is no end to the variety of architectures that can be created to fulfill them. At the same time, although functionality is independent of any particular structure, it is achieved by assigning responsibilities to architectural elements.

If functionality were the only thing that mattered, systems would'n have to be structured in pieces or components at all; a monolithic blob without internal structure would do the work fine [103]. Instead, systems are designed as structured set of collaborative architectural elements to induce non-functional properties on them, which are modernly referred to as *quality attributes*.

A *quality attribute* (QA) is a *measurable or testable property of a system that is used to indicate how well the system satisfies the need of its stakeholders*. In that sense they

measure the “goodness” of a product along some dimension of interest to a stakeholder. In effect, systems are frequently redesigned not because they are functionally deficient, but because they are difficult to maintain, port, scale, or are too slow. These mentioned attributes are a few examples of QAs of a system (*modifiability, portability, scalability* and *performance*). More examples include: *availability, interoperability, security, testability* and *usability*. Some others that arise frequently are: *variability, performance, mobility* and *monitorability*.

3.3 System requirements and SA

System requirement encompass the following categories: *functional requirements, quality attribute requirements* and *constraints*.

Functional requirements state what the system must do, and how it must behave or react to runtime stimuli. They are satisfied by assigning an appropriate sequence of responsibilities to architectural elements, which is a fundamental design decision for a software architect.

QA requirements are qualifications of the functional requirements or of the overall product. They are satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures [103].

A *constraint*, in the context of an architecture, is *a design decision with zero degrees of freedom*. That is, it’s a design outcome that exists prior to the design process starting. Constraints usually are motivated by external factors, organizations’ internal rules and stakeholders’ particular interests. Some examples include the requirement to use an specific programming language, or to reuse a certain existing software module. They are satisfied by accepting the design decision and reconciling it with other affected decisions.

3.4 Network based systems and peer-to-peer architectures

A network-based system is that where communication between components is restricted to message passing [110], or to an equivalent technique if a more efficient mechanism can be selected at run-time based on the location of components [111]. Network-based systems are those capable of operation across a network, but not necessarily in a transparent fashion to the user. It could be desirable and sound for the system user to be aware of the difference between an action that triggers a network request and another that can be responded locally. That increases its value in scenarios where network usage implies an extra transaction cost.

Network-based systems architectures can be cataloged into two big groups: centralized or vertical and decentralized or horizontal architectures. The key characteristic of vertical architectures is that there are always some components that perform the main processing tasks. The mission of those components is to wait for another component's request to build an answer and send it back. This way some components are classified as *servers* and others as *clients*.

In decentralized systems, also known as peer-to-peer, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load. From a high-level perspective, the processes that constitute a peer-to-peer system are all equal. Each of them is considered equally important in terms of initiating an interaction. That means that every process that constitutes the system represents functions that need to be carried out. Consequently, much of the interaction between components is symmetric: each of them will act as a client and a server at the same time. Peers can be added and removed from the peer-to-peer system with no significant impact [112] [103].

3.5 Architectural tactics, patterns and styles

The design of a software system consists of a collection of decisions, some of them impacting in the system's functionalities and other in the responses of QAs. An *architectural tactic* is a design decision that influences the achievement of a QA and affect the system response to some stimulus. *Tactics* are the building blocks of design and have been used by architects by years. They help designers to construct a system fragment from "first principles".

Some tactic examples are: *manage resources* to improve *performance*, *reduce size of a module* and *increase cohesion* to improve *modifiability*, *detect faults* to keep *availability* and *limit complexity* to increase *testability*.

Tactics have been cataloged by several researchers [103] [113] [114], but they can overlap, been possible to face with multiple options as candidates for optimizing an specific QA. At the end, the application of a tactic depends on the targeted context. An example of that idea exposed in [103] is that the tactic *manage sampling rate* to improve *performance* is relevant in some real-time systems, but not in all.

Since to produce a successful architectural design is complex and challenging, designers have been looking for ways to capture architectural knowledge and to reuse well-proven

designs. In some cases, architectural elements are composed in ways that solve particular problems. Over time, the composition has been found useful over different domains, and so they have been documented. These well-known and reused compositions of architectural elements are called *architectural patterns* and they are, by definition, discovered in practice.

The *pattern* definition includes a *context* representing a common situation in the world that gives rise to a problem, a *generalized problem* that arises in the given context and an *architectural solution* to that problem. A pattern is described as a solution to a class of problems in a general context. When a pattern is chosen and applied, the context of its application becomes very specific [103]. As it is reinforced in [103]: “*there will never be a complete list of patterns: patterns spontaneously emerge in reaction to environmental conditions, and as long as those conditions change, new patterns will emerge*”.

Patterns make possible to design an architecture for a software system without starting from scratch. Architects usually select, tailor and combines patterns. They must decide how to make them fit with a particular context and the environment’s constraints. When no candidate patterns are available for an specific domain or problem, architects still can get hands on basic architectural tactics. In principle, a tactic is focused on a single QA—i.e., a tactic does not considers trade-offs, they must be considered and controlled explicitly by the architect. That is a key difference with respect to patterns: trade-offs are built in into patterns. Most patterns are constructed from several tactics, which are chosen to promote different QAs.

When an architectural pattern is visualized isolating it from its context, and just focusing on the proposed structure and the interaction between its software elements, it is called an *architectural style*. An *style* is a *coordinated set of architectural constraints that restrict the roles of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style* [102].

Styles are a mechanism for categorizing architectures and for defining characteristics common to each category. Each style abstracts the interactions between components, that way capturing the essence of a pattern of interaction by ignoring the details [115]. According to Perry and Wolf [108] an *architectural style* is *an abstraction of element types and formal aspects from various specific architectures, perhaps concentrating on only certain aspects of an architecture*. An style encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships.

When designing a network-based application, a software architect must understand the

problem domain and thereby the communication needs of the application components. Besides, he should be aware of the variety of architectural styles and the particular concerns they address. It is under his responsibility to choose the right architectural styles and to anticipate the sensitivity of each interaction style to the characteristics of the application.

3.6 Network-based architectural styles

This section presents a survey of common architectural styles that have been used by the designers community when building network-based applications. Of course, the list is by no means the set of all possible network-based application styles. In fact, a new style can be born from the addition of an architectural constraint to any other style.

3.6.1 Pipes and Filters

In this style, each *component* is called a *Filter*, and it reads streams of data on its inputs and produces streams of data on its outputs. Normally, a *Filter* applies a transformation to its input streams and processes them incrementally—i.e, output begins before the input is completely consumed. This style is also referred to as a one-way data flow network. The constraint is that a filter must be completely independent of other filters [104].

Pipes and Filters style enables simplicity, as it allows the designer to understand the overall input/output of the system as a simple composition of the behaviors of individual filters. It also supports reusability, as any two filters can be hooked together, provided they agree on the data that is being transmitted between them. Finally, structures based on this style can be easily maintained and enhanced due to the fact that new filters can be added to existing systems.

Figure 3.1 – Pipes and Filters style (adapted from [104])

Some disadvantages of the style are: a propagation delay is added through long pipelines, batch sequential processing occurs if a filter cannot incrementally process its inputs, and no interactivity is allowed. Another disadvantage is that a filter cannot interact with its environment because it cannot know that any particular output stream shares a controller

with any particular input stream. These properties decrease user-perceived performance if the problem being addressed does not fit the pattern of a data flow stream.

3.6.2 Replicated Repository

This style suggests to improve the *accessibility* of data and *scalability* of services by having more than one process providing the same service [110]. These processes interact to provide clients the illusion that there is just one centralized.

Primary advantage of *Replicated Repository* is the improvement of user-perceived performance, both by reducing the latency of normal requests and enabling disconnected operation in the face of failures. *Simplicity* remains neutral, since there is a complexity on doing the replication. Maintaining *consistency* is the primary concern.

3.6.3 Shared Repository

Share Repository style sets a central data structure on which independent components operate. This way, interaction between components occurs indirectly. The data structure can be used to store current state of some task inside the system. Interactions between the repository and its external components can vary significantly between systems.

This style presents variants or subcategories depending on role of the central repository. An special and well-know variant is called as *Blackboard*. In this case, the current state of the central data structure is the main trigger of selecting processes to execute [104].

Figure 3.2 – Blackboard style (adapted from [104])

Blackboard systems have been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition.

3.6.4 Cache

This style can be considered a variant of *Replicated Repository*. It consists on the replication of the result of an individual request such that it may be reused by later requests. This form of replication is most often found in cases where the potential data set far exceeds the capacity of any one client, or where complete access to the repository is unnecessary. It can be implemented in two strategies: lazy or active replication.

Lazy replication occurs when data is replicated upon a not-yet-cached response for a request, relying on locality of reference and commonality of interest to propagate useful items into the cache for later reuse. On the other hand, pre-fetching cacheable entries based on anticipated requests is called active replication [102].

Cache style provides slightly less improvement than the replicated repository style in terms of user-perceived performance, since more requests will miss the cache and only recently accessed data will be available for disconnected operation. On the other hand, caching is much easier to implement, due to it does not require as much processing and storage. At the same time, it is more efficient, because data is transmitted only when it is requested.

3.6.5 Client-Server

The *Client-Server* style is the most frequently encountered in the design of network-based applications. A *server* component, offering a set of services, listens for requests. A *client* component, desiring to use a service, sends a request to the server via a *connector*. The *server* either rejects or performs the request and sends a response back to the *client* [102].

According to Andrews [110] a *client* is a triggering process; a *server* is a reactive process. *Clients* make requests that trigger reactions from *servers*. A client initiates activity at times of its choosing; it often then delays until its request has been serviced. By its part, a server waits for requests to be made and then reacts to them. A server is usually a non-terminating process and often provides service to more than one client.

Aiming to improve scalability the server component must be simplified by a proper separation of functionality. This simplification usually takes the form of moving all of the user interface functionality into the client component, allowing the *client* and *server* to evolve independently.

Some variations of *Client-Server* exists, for instance, the *Client-Stateless-Server* style,

Figure 3.3 – Client-Server architectural style

which add the constraint that no session state is allowed on the server component. Each request from *client* to *server* must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is kept entirely on the client. These constraints help to improve *visibility*, *reliability*, and *scalability* [102].

Another variation is the *Remote Session* style, which attempts to minimize complexity and increase reusability of client components. Each *client* initiates a session on the server and then invokes a series of services on it, exiting the session at the end. Application state is kept entirely on the server. This style is commonly to access remote services by means of a generic *client*. Advantages are that it is easier to centrally maintain the interface at the server, reducing concerns about inconsistencies in deployed clients when functionality is extended. Efficiency is also improved since interactions make use of extended session context on the server. Disadvantages are that it reduces *scalability* of the *server* and reduces *visibility* of interactions.

The *Remote Data Access* style [116] is yet another variant of *Client-Server* that spreads the application state across both client and server. Its advantages are that a large data set can be iteratively reduced on the server side without transmitting it across the network, improving *efficiency*. At the same time, *visibility* is improved. Some disadvantages are lost of simplicity and decrement of scalability, since application context is stored on the *server*.

3.6.6 Layered System and Layered-Client-Server

The *Layered System* style organizes an application hierarchically. The system is broken in a sequence of adjacent layers where each of them provides services to the layer above it and uses services from the layer below it [104].

Figure 3.4 – Layered System architectural style

Layered System style improves *evolvability* and *reusability*. Its application reduces the coupling across multiple layers as it hides inner layers from all except the adjacent outer layer. The primary disadvantage of the style is the addition of overhead and latency to the processing of data, reducing user-perceived performance [102].

Although *Layered System* is considered a “pure style”, its use within network-based systems is limited to its combination with the *Client-Server* style to conform the *Layered-Client-Server* style [102]. Architectures based on *Layered-Client-Server* are referred to as two-tiered, three-tiered, or multi-tiered architectures [116].

3.6.7 Code on demand

In this style a client component has access to a set of resources, but it does not have the know-how on how to process them. It sends a request to a remote *server* for the code representing that know-how, receives that code, and executes it locally [117].

Advantages of the style include the ability to add features to a deployed client, which improves *extensibility* and *configurability*. At the same time it increases the user-perceived *performance* and *efficiency* when the code can adapt its actions to the client’s environment and interact with the user locally, rather than through remote interactions [102].

On the other hand *simplicity* gets reduced because of the need to manage the evaluation environment. *Scalability* of the *server* component is improved, since it can off-load work to the client that would otherwise have consumed its resources.

3.6.8 Remote Evaluation

Remote Evaluation is also derived from the Client-Server style, a *client* component has the know-how necessary to perform a service, but lacks the computational resources required (CPU, data source, etc.). Those resources are located at a remote site. The *client* sends then the know-how to a *server* at the remote site. In turn, the *server* executes the code using the resources available there. The results of that execution are then sent back to the *client* [117].

Some advantages of the style include the ability to customize the services of the *server*, which improves *extensibility* and *customizability*. More *efficiency* is also targeted as the code can adapt its actions to the environment inside the server. *Simplicity* is reduced due to the need to manage the evaluation environment. *Scalability* is reduced as well.

3.6.9 Mobile Agent

This style can be considered a derivation of Remote Evaluation and *Code on Demand* styles. An entire computational component is moved to a remote site, along with its state, the code it needs, and possibly some data required to perform the task [117].

The primary advantage of the style is the dynamism in the decision on when to move the code. An application can be in the middle of processing information at one location when it decides to move to another location, presumably in order to reduce the distance between it and the next set of data it wishes to process. Therefore, the reliability is improved because the application state is in one location at a time [117].

3.6.10 Implicit Invocation

Implicit Invocation reduces coupling between components by removing the need for identity on the *connector* interface. Instead of invoking another component directly, a component can broadcast one or more events. Other components can register interest in that type of event and, when the event is announced, the system itself invokes all of the registered components [104].

This style supports *extensibility* thanks to the option and easy of adding new components that listen for events, *reusability* by encouraging a general event interface and integration mechanism, and *evolvability* by allowing the replacement of component without affecting other components [104].

Implicit Invocation improves efficiency in applications that are dominated by data monitoring, rather than data retrieval, since it eliminates the need for polling interactions. According to Fielding [102] the basic to implement the style is to build one event bus to which all components listen for events of interest to them. That can lead to *scalability* issues with regard to the number of notifications and a single point of failure in the notification delivery system. According to the author, all that can be ameliorated though the use of Layered Systems styles and filtering of events, at the cost of *simplicity*. The style can be also accused of low *understandability* as far as it could be hard to anticipate what will happen in response to an action.

3.6.11 C2

C2 style [111] combines *Implicit Invocation* with *Layered-Client-Server*. It sets as the sole means of components communication the interchange of asynchronous notification messages going down, and asynchronous request messages going up. This enforces loose coupling of dependency on higher layers and zero coupling with lower levels, improving control over the system without losing most of the advantages of *Implicit Invocation* [102]. Notifications are announcements of a state change within a component, not been constrained what should be included with a notification.

A connector's primary responsibility is the routing and broadcasting of messages; its secondary responsibility is message filtering. Problems related to scalability that are present in *Implicit Invocation* are reduced by the introduction of layered filtering of messages, improving *evolvability* and *reusability* as well. All that makes that *C2* is directed at supporting large-grain reuse and flexible composition of system components.

3.6.12 Distributed Objects

The main characteristic of the *Distributed Objects* style is that it organizes a system as a set of components interacting as peers. An *object* is defined as an entity that encapsulates private state data, a set of operations that manipulate that data, and possibly a thread of control. All those elements can be considered a single unit [118].

An objects state is completely hidden from all other objects and there is a single way that state can be examined or modified: by making a request to the object or invoking one of the object's publicly accessible operations.

The former restriction improves *evolvability*, since it creates a well-defined interface for each object, enabling the specification of an object's operations to be made public while

keeping the implementation of its operations and the representation of its state information private [102]. Application state is distributed among the objects, what represents a benefit in terms of keeping the state where it is most likely to be updated, but has a disadvantage in the sense that it decreases the *visibility* of the overall system activity.

Object systems are designed to isolate the data being processed. As a consequence, data streaming is not supported in general, what provides better support for object mobility when combined with the *Mobile Agent* style [102]. A drawback of this style is that, in order for one object to interact with another, it must know the identity of that other object. Therefore, if the identity of an object changes, all other objects that explicitly invoke it need to be altered [104].

3.6.13 Brokered Distributed Objects

Aiming to nullify the object identity issue, modern distributed object systems use one or more intermediary styles to facilitate communication. This includes event-based or Implicit Invocation integration and brokered Client-Server [119]. The *Brokered Distributed Objects* style introduces *name resolver* components whose purpose is to answer client object requests for general service names with the specific name of an object that will satisfy that request.

Although that technique improves significantly QAs as *reusability* and *evolvability*, the extra level of indirection requires additional network interactions, reducing *efficiency* and *performance*.

Brokered Distributed Objects applications are most used in applications that involve the remote invocation of encapsulated services, such as hardware devices, where the efficiency and frequency of network interactions is not a key concern [102].

3.7 Software Design Patterns

The other activities included in the software design process, that are not considered architecture design, have as target elements of lower level of abstraction. They are called *design patterns*, and are focused in concerns like the composition of data structures, classes and the collaborations between them, user interfaces, etc. A design pattern is defined as an important and recurring system construct.

The use of design patterns and pattern languages to describe recurring abstractions in object-based software development have been explored by the Object-Oriented Program-

ming (OOP) community. Patterns's design space includes implementation concerns specific to the OOP techniques, such as class inheritance and interface composition.

A pattern language is a system of patterns organized in a structure that guides application of those patterns [120]. A compendium of software design patterns is presented and described in [121].

In general, a pattern, or pattern language, can be thought of as a recipe for implementing a desired set of interactions among objects. A primary benefit of patterns is that they can describe relatively complex protocols of interactions between objects as a single abstraction [122]. In other words, a pattern defines a process for solving a problem by following a path of design and implementation choices [123].

4 DESIGNING A RTO SYSTEM

An RTO system's major goal is to lead the plant to the best operational conditions the system can “figure out”. The intended end-users of the system are the plant operators and people that take decisions about where to drive the production to.

One of the challenges of this thesis was to elicit which are the features an RTO system should have, in order to be relevant for industry and academic area as well. That task was performed taking into account what the reviewed RTO approaches have in common, which are common practices in industry and some academic interests. All them were expressed then in terms of functional and quality attributes requirements and constraints.

4.1 RTO Application domain requirements

4.1.1 Functional requirements

- FR-1** Read the desired state of affairs from external sources specified as optimization targets.
- FR-2** Extract data at real time from the process allowing the user to define which variables will be sampled, as well as the samples' time window. Data can belong to most recent plant state, last n states or n states from a past time period. The system should allow the user to visualize graphically the sampled data as temporal series.
- FR-3** Analyze the process based on its snapshot and get to conclusions about its state: transient or steady-state. In case of the process been in stable, the system must build a representational set of values for the sampled variables. This activity must be done in parallel with the sampling one described in **FR-1**. Besides an assertion about actual process state, the system should calculate some interesting indexes and variables—e.g actual profit.
- FR-4** Keep a historical record of the process states classified as steady-state, allowing the user to consult it at any time. The user must also have te possibility to limit that record to the last m elements.
- FR-5** Allow the implementation of several RTO approaches, without needing to stop the system to change the approach, been possible to run more than at the same time. The system should provide some mechanism to benchmark the approaches in order

to choose which of the suggested optimal operational conditions will be finally passed to the control layer as set-points updates.

- FR-6** Keep a historical record of the optimal operational conditions suggested by the RTO approaches and which of them were applied to the plant. The user must be able to consult it at any time and also to limit the record size to the last m elements.
- FR-7** Communicate at real time with the plant control layer to send optimal operational scenarios as set-point updates. Actual process conditions must be taken into account before sending updates. In this sense, the RTO system must check if suggested operational scenarios are feasible to be set as targets.
- FR-8** Allow the implementation of the RTO cycle in open-loop or close-loop, been the user's decision which of both strategies to apply.

4.1.2 Quality attribute requirements

- QAR-1** *Usability*: It must be relatively easy to switch from one RTO approach to other while using the system or to run several in parallel, even to develop and add new approaches.
- QAR-2** *Variability*: It must be relatively easy to implement RTO concerns inside the system, as well as to extend or override the already implemented. As different positions exist in the academic field about how to implement some concerns (e.g., doing data reconciliation and parameters estimation together or independently), the system must provide technical spots to do handle that.
- QAR-3** *Interoperability*: The system must be capable to interpret with quality all meaningful information exchanged through the interfaces designed to communicate with the plant systems.
- QAR-4** *Modifiability*: The interfaces implemented to communicate with the plant, both for sampling and set-points update, must be sufficiently generic. The system must offer a relatively easy mechanism to adapt those interfaces in order to cover the variety of technology used by actual industry.
- QAR-5** *Performance*: The system structures to be designed cannot add unnecessary performance costs to those that are inherent to RTO steps and network interactions between components and with the plant. Interaction styles should be chosen carefully to handle performance issues. Properties like operations latency should be observed to avoid decrement of user-perceived performance.

- QAR-6** *Testability*: The software prototype to be developed should be easy to test, as individual functional units or as a whole, been feasible to made the system to demonstrate its faults.
- QAR-7** *Portability*: It must be relatively easy to change the software in order to run it in another platform (Operative System).
- QAR-8** *Scalability*: The system must be designed in a way that it is easy to scale hardware resources used by the application. For example, if the optimization step of RTO is taking too long, the hardware where that code is running can be scaled without notable efforts.
- QAR-9** *Monitorability*: Operations carried out by the system should be easy to monitor by the operations staff.

4.1.3 Constraints

- C-1** In the case of model-based RTO approaches, models must be equation oriented.
- C-2** To use EMSO [124] as simulation, optimization and parameter estimation engine.
- C-3** To use free and open-source software technologies during development.

4.2 Architecture design process

This section provides a general overview of the RTO system architecture by walking through the process of deriving it. The structures designed as part of the architecture, and the behaviors and interactions of the elements that populate those structures, were derived from the application of well known architectural styles. Throughout the design, functional requirements were satisfied by assigning an appropriate sequence of responsibilities to each architectural element. Quality attributes requirements were approached by the properties that architectural styles induce in the system. Constraints were satisfied by accepting, reconciling and integrating them with other related design decisions.

There exists two common perspectives on how to approach the process of software architectural design. The first says that the designer starts with nothing —a blank slate or drawing board — and builds-up an architecture joining familiar components until it meets expectations for the intended system. The second perspective sets that the designer starts with all the system functionalities as a whole, without internal architectural constraints, conforming a monolithic component. Then, he incrementally identifies and applies constraints to the system elements aiming to create differentiation into the design. That differentiation creates the forces that define system behavior to flow naturally and in

harmony with the rest of the elements. Where the first perspective emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context [102]. The software architecture proposed in this dissertation for an RTO system has been developed using the latter process.

As new architecture styles are applied and decisions are made at each design iteration, abstract concepts turn into more concrete software components and functionalities get more localized. Each new architectural component is less abstract than its predecessors. Components that are concrete at some point turn into more abstract ones in the next iteration, giving rise to new components with a higher level of specification.

4.2.1 First design iteration: starting from the Null Style

In terms of software architecture, the null style is equivalent to a bird eye view of a system in which there are no distinguished boundaries between components. At this level, the RTO system can be described as a monolithic architectural component that interacts with its environment through three ports. First port is used to read the desired state of affairs in terms of optimization targets. That information is generated by the Planning and Scheduling layers of the plant decision hierarchy presented in figure 2.1. Second port is used to sample and query the plant in order to obtain data about its actual and past state. Through third port, the RTO system sends proposals for updating control set-points aiming to lead the plant to a better operational condition.

Figure 4.1 – Starting point of the RTO system architecture design

There is a computational process running inside this single component that links the three architectural ports semantically. That process is all about finding out a solution, every time it is needed, for the RTO central problem. As other architecture levels and components are described, the way that process gets implemented comes out.

4.2.2 Second design iteration: a layered system

First constraints added to the system design came from the Layered System architectural style. As described in section 3.6.6, that style conceives a system as been composed of hierarchical layers, constraining behavior in a way that each component cannot “see” beyond the immediate layer it interacts with.

At this level, the RTO system can be described as two layers organized in an horizontal hierarchical layout, each of them containing one software component. The higher layer in the hierarchy, identified in figure 4.2 as OPTIMIZATION and represented by its internal component OPTIMIZER, has the responsibility of figuring out candidates set-points updates. It will trigger RTO cycles requesting services from lower layer, identified as PLANT INTERFACING and instantiated by PLANT DRIVER component. According to the applied architectural style’s restrictions, only PLANT INTERFACING layer, and by consequence, its internal component, will be in direct contact with the plant. PLANT DRIVER implements all the functionalities that have to do with interfacing the plant and analyzing its state, including sampling the process, taking a position about its actual state (transient or steady-state) and communicating with control structures. It is always ready to serve OPTIMIZER’s requests.

Figure 4.2 – RTO system architecture after applying Layered System style

The application of the Layered System style contribute to some quality attributes requirements identified in section 4.1.2. By restricting components to interact with a single layer, a bound on the overall system complexity is set and components independence is promoted. OPTIMIZER and PLANT DRIVER can evolve independently. Separation of concerns is also enabled. By separating plant interfacing actions from optimization operations, portability of the RTO system is improved as multiple technological scenarios

can be approached. Scalability is also benefited as OPTIMIZER component is simplified.

4.2.3 Third design iteration: first distributed objects

Next step in the design process added internal architectural constraints to OPTIMIZER and PLANT DRIVER components, searching for clarity on how their operations occur how to cover functional requirements.

Designing PLANT DRIVER

Design principle of *separation of concerns* led the decisions taken over PLANT DRIVER component. Three concerns were identified according to functional requirements described in section 4.1.1. The first of them, called here as PD-C1, touches on functional requirement RF-2. PD-C1 induced to think in the need of having a permanent information service providing on-demand data, containing sampled values of process variables for some time interval. Requests could ask for most recent values of variables or past ones.

Second concern is represented by functional requirements RF-3 and RF-4, and also motivated to have a similar service providing information about the process operational state. Delivered data must include a predicate saying if the process is stable or in transient state at real time. In case of an stable process, a representational set of values must be provided as part of the serviced data. The service must keep a record of past identified steady-states. Other interesting indexes and variables about actual process state can be calculated as part of the analysis run. This concern will be label as PD-C2.

Third concern, labeled PD-C3, is linked to requirements RF-7 and RF-8. It gets summarized in the idea that there is a clear need for implementing a software interface for communicating with plant control structures, and to deal with details that communication brings up. As RF-7 set clear, the RTO system must also provide some feasibility checking mechanism running as a last step before sending set-point values to the control layer.

With those concerns in mind, responsibilities of PLANT DRIVER component were split, deriving in two internal subcomponents. Each of them acts as a distributed object—i.e., as an independent entity. As concerns PD-C1 and PD-C2 are related to each other, both were covered by first subcomponent, identified in figure 4.3 as PROCESS INFORMATION SERVICE. Is this software unit the one that will provide an information service offering data about plant variable values and plant operational states.

Figure 4.3 – RTO system architecture after PLANT DRIVER design

Concern PD-C3 was covered by subcomponent PROCESS CONTROL INTERFACE. This software unit has the mission to serve as an interface with the plant control system and take care of which are the final set-point updates that will be suggested to that system. In order to fulfill its mission, in addition to getting communication with control softwares, the subcomponent uses services provided by PLANT INFORMATION SERVICE.

Designing OPTIMIZER

Functional requirements RF-1, RF-5 and RF-6 were sources of the concerns approached in the internal design of OPTIMIZER component. Quality attributes requirements mentioned in section 4.1.2 influenced responsibility allocations and design decisions as well.

The RTO system must behave as an entity that is always led by a desired state of affairs, specified to it by some human or software entity, in the form of optimization targets. That is why, a key concern called here as OP-C1, is the real-time suggestion of best operational plant states. Suggestions will be a derivation from optimization targets, actual conditions and used approaches. Being so, at time t , OPTIMIZER uses an approach a to idealize a future scenario $S_{a,t}$, where the plant will be fulfilling optimization targets. In terms of plant operation, $S_{a,t}$ can be defined as a pair (V, i) , where V represents a snapshot of several process variables' values, and i represents a quantifiable improvement to the process. Whenever OPTIMIZER triggers a new iteration of approach a , its final target will be to craft an $S_{a,t}$, promising an improvement i if process variables hit values specified in V . As reviewed literature points, there are several options for a and how to determine V , each of them with different principles and algorithms as foundations. That

is why, OPTIMIZER must endow the RTO system of the ability to use several approaches.

Another concern is also very well delimited. As required in RF-5, a decision mechanism must exist inside OPTIMIZER to pick an optimal operation scenario if more than one was suggested. That is motivated by the fact that only a single set V can be translated into set-points and applied to the plant at a time. Let's identify that concern as OP-C2.

After analyzing concerns, component OPTIMIZER was split into two independent objects. First of them is responsible for covering concern OP-C1, and was named IMPROVER. As described, its mission is to read and interpret optimization targets and turn them into scenarios of improved operation. Internal computational process that runs inside IMPROVER requests information to the neighbor layer, specifically to PROCESS INFORMATION SERVICE component. Based on that information, and using several approaches, it proposes scenarios to the second component, named ACTUATOR, through an architectural connector of type Shared Repository.

Figure 4.4 – RTO system architecture after OPTIMIZER design

Main roll of ACTUATOR is to choose which scenario will be translated into set-points update. To achieve that, this component will run some decision making process, based on consistent criteria. From now on, that action will be referenced as the Pick Out Procedure (POP).

ACTUATOR also creates a natural spot for adding to POP a weighting based criterion,

to be used when received (V, i) pairs are derived from different optimization targets read by IMPROVER. That is, when several targets are approached at the same time, one of them will prevail when the time comes to pick out a set-points set.

POP will consist of two phases:

Phase I (POP-DNF): discarding non-feasible candidates. This phase consists in the application of filters to the candidate solutions to discard those that are not feasible to be applied, for some reason. Each filter serves as a check for an specific criteria. The one implemented in the prototype calculates a value indicating how far the candidate solution is from the actual operational conditions, but many other could be implemented and run in sequence, as a chain.

Phase II (POP-CB): choosing the best candidate from the list of feasible ones.

Several strategies could be used by ACTUATOR in order to trigger POP. Follows some examples:

- First available: POP will be started every time a new candidate is deployed at the repository. In this case, just the POP-DNF phase will occur.
- Timer starting on first available: ACTUATOR will start a configurable timer to give a chance to other candidates be deployed. Expired the time, POP starts considering the candidates present in the repository at that time.
- Frequent checks: ACTUATOR makes frequent checks to the repository, and starts POP every time it finds at least one candidate deployed.

In order to help itself to perform POP, ACTUATOR can consume services from PROCESS INFORMATION SERVICE. It will also keep a record with all scenarios proposed along time, and which of them were selected and why.

4.2.4 Fourth design iteration: deeping into PROCESS INFORMATION SERVICE

Next architectural design iteration dealt with how to structure PROCESS INFORMATION SERVICE component. Considering that system services derived from concerns PD-C1 and PD-C2 should run concurrently, two separated components were introduced. First of them, identified as DATA INGESTOR, takes ownership of data sampling related tasks. It must give to system users the possibility to configure how to perform sampling

tasks, in terms of frequency and process variables.

DATA INGESTOR is a vital component, once the RTO system is a data intensive process. Feeding the system with good data is crucial for a successful behavior of it. Data passed to other system layers or components should be treated before. Being so, DATA INGESTOR endows the RTO system with a natural place where to address data treatment concerns like gross errors detection and data reconciliation. Pipe and Filters architectural style described in section 3.6.1 can be applied to address those data treatment actions.

Second component, identified as ANALYZER, runs a periodic analysis process. At each iteration, it runs algorithms and methods in order to figure out if plant is stable or not. If that is the case, a representative state composed of several variable values will be available. Users must have options for configuring the analysis in terms of frequency, process variables considered as relevant for steady-state identification, and detection methods.

Figure 4.5 – Architectural view of PROCESS INFORMATION SERVICE component

As is clear here, this component is a natural spot for using steady-state detection methods produced by industry and academic fields. The possibility to run more than one method in parallel is not discarded, so ANALYZER could be provided of some decision taking mechanism helping it to claim its final predicate about plant stability. Due to almost every steady-state detection method uses a time window of data to perform its analyses, ANALYZER and DATA INGESTOR communicates indirectly using a connector of type Shared Repository. In this case, DATA INGESTOR acts as a data provider and ANALYZER as a consumer. The connector is responsible for offering users options for setting

the repository behavior (e.g, its capacity).

Pipes and Filters (section 3.6.1) architectural style could be applied to design internal structures of DATA INGESTOR component, in order to tackle key RTO concerns as Gross Errors Detection and Data Reconciliation. Filter components could input data read from the plant and apply tests to verify each received sample. Those samples containing some erroneous tag value could be discarded or fixed, conciliating it with other values.

4.2.5 Fifth design iteration: deeping into IMPROVER

Pursuing to shed more light on how IMPROVER component do its work, next design iteration addressed how is it internally structured and which mechanism is used to communicate with ACTUATOR.

Plurality of RTO approaches described in RF-5 was covered allowing to have, at running time, several instances of IMPROVER component running as concurrent entities. All instances will behave as autonomous components targeting to suggest better operational scenarios. Each instance will run a RTO approach and, in principle, could receive different optimization targets. For simplicity, just two instances of IMPROVER were represented in figure 4.6, identified as IMPROVER A and IMPROVER B.

Figure 4.6 – Architectural view of IMPROVER component

Every time an instance has a candidate scenario, it will be staged in the repository. ACTUATOR gets aware of it and starts POP according to one of its strategies.

5 PREPARING THE WORKBENCH

At this point, a software architecture has been proposed for an RTO system with requirements specified in section 4.1. The architecture, presented from a structural view of components and connectors, tries to enable properties that form a superset of the requirements. It is time now to define how to approach the implementation of a system prototype. For that sake, this chapter touches on the paradigms, standards and software technologies selected to be used in the development. Next two chapters will describe how all pieces were put together in order to implement running systems.

5.1 Standars and Paradigms

5.1.1 Software Agents

As already mentioned, RTO systems keep a long-term interaction with the plant, so they are better described in terms of their on-going behavior. That lead us to classify them as *reactive systems* and to associate their nature with an special subclass of reactive systems. Software programs from that class are called *Agents*.

Software Agent is a concept that comes from the Artificial Intelligence (AI) field. An *Agent* is a reactive system situated within and a part of an environment, that senses that environment and acts on it continuously over time, pursuing its own agenda with well defined objectives [125]. *Agents* have some degree of autonomy in the sense that targets are delegated to them to be fulfilled and they determine how best to achieve it [126].

A definition for the term *software agent* has been addressed in several works ([127] [128] [129] [130]). In general, all the definitions converged to the idea that an agent is essentially *a special software component situated within and a part of an environment that senses that environment and acts on it over time; that has autonomy or behaves like a human agent, working for some clients in pursuit of its own agenda with social or communication abilities*. All software agents are programs, but not all programs are considered agents.

A software agent is defined as autonomous because it operates without the direct intervention of humans or others and has control over its actions and internal state. It is considered to have a social character because it communicates and cooperates with humans or other agents in order to achieve its tasks.

An agent perceives its environment and responds to changes that occur in a timely fashion. That is why it is reactive. At the same time an agent has a proactive behavior due to it does not just act in response to its environment, but is able to take the initiative and to exhibit a goal-directed behavior.

Figure 5.1 – An agent in its environment (after [5])

Providing the certainty that it will not deliberately communicate false information an agent can be truthful. It can be benevolent always trying to perform what is asked of it. It also can be rational, always acting in order to achieve its goals and never to prevent its goals being achieved. Besides, a software agent can learn, adapting itself continuously to fit its environment and to the intention of its users. If necessary and if the environment and the computational infrastructure permits, an agent can be also mobile with the ability to travel between different nodes in a computer network. [131] provided a general taxonomy of software agents.

As an agent is a software entity, an architecture can be defined for its internal organization. First efforts in the agent-based computing domain focused on the development of intelligent agent architectures. More recently several established styles of architecture for a software agent have been documented [132]. These architectures can be divided into four main groups: logic based, reactive, Belief-Desire-Intention (BDI) and layered architectures.

Logic-based architectures have their foundation in traditional knowledge-based systems techniques in which an environment is symbolically represented and manipulated using reasoning mechanisms. Reactive architectures implement decision-making as a direct mapping of situation to action and are based on a stimulus-response mechanism triggered by sensor data.

BDI architectures can be considered the most popular agent architectures [133]. Their origins come from philosophy and present a logical theory based on the mental attitudes

of belief, desire and intention using a modal logic.

Agent Oriented Programming

In 1993, Yoav Shoham proposed a new programming paradigm which he called Agent-Oriented Programming (AOP) [134]. The paradigm was based on a societal view of computation. AOP provides the fundamental idea of programming software agents in terms of mentalistic notions that are used to explain human behaviors (such as beliefs, desires, and intentions). That proposal is based on the facts that humans use such concepts as natural abstraction mechanisms for representing the properties of complex systems [5].

The AGENT0 programming language [135] was the first implementation of the AOP paradigm. In this language, an agent is specified in terms of a set of capabilities (things the agent can do), a set of initial beliefs, a set of initial commitments, and a set of commitment rules. The commitment rule set is what determines how the agent acts. Each rule includes a message condition, a mental condition, and an action. The message condition is matched against the messages that the agent has received aiming to determine whether such a rule can be triggered. The mental condition is matched against the beliefs of the agent. If the rule fires, then the agent becomes committed to the action [5].

AgentSpeak(L) [136] is another agent-oriented programming language that is based on logic programming and the BDI architecture. It became more popular as AgentSpeak. In recent years, the language has been used and improved in research initiatives, resulting in extensions and formal verifications of AgentSpeak programs. The language is also one of the most popular agent-oriented languages because of the development of the Jason platform [126].

Agent-Oriented Software Architecture (AOA) is also a relatively new software paradigm that bring concepts from AI into the domain of distributed systems. According to the AOA approach, a software application can be modeled and built as a collection of interacting agents.

As mentioned in the beginning of this section, after reviewing the literature and confirming similarity between software agents behavior and that of RTO programs, a decision was made: to use agents for implementing those architectural components where the reactive character is the principal attribute.

Multi-Agent systems

An agent system may be based just on a single agent, but more practical implementations show multiple agents interacting. A Multi-Agent System (MAS) is one that consists of a number of agents working and interacting with each other within an environment. The interaction is usually performed exchanging messages through some computer network infrastructure [5].

According to [137], MAS and AI are corner-stone frameworks to design an intelligent real-time system for complex industrial facility management and control. MAS have found application in the industry since the late 80's, but no reference has been found to its specific application in on-line optimization environments.

The typical structure of a MAS is illustrated in figure 5.2. The represented system shows a number of agents communicating with one another. They are acting in an environment. Each agent can act or have control in an specific region of the environment. These *spheres of influence* may overlap, what may give rise to dependencies between the agents.

Figure 5.2 – Canonical view of an agent-based system (taken from [138])

Complex systems can be modeled as MASs, having the agents common or conflicting goals. In order to fit their goals agents may cooperate for mutual benefit or compete to fulfill their own purposes. The interaction between them can happen both indirectly (by acting on their environment) and directly (communicating and negotiating with each other) [132].

When dealing with a MAS, it cannot be assumed that the agents are all aligned in terms of intentions and goals. Generally agents will be acting on behalf of users or owners with very different goals and motivations. That is why, in order to successfully interact, they require the capacity to communicate, cooperate, coordinate, and negotiate with each other [5].

A variety of application fields are being benefited with the use of multi-agent systems. Examples of industrial domains where they have been fruitfully employed include process control [139], system diagnostics [140], manufacturing [141], transportation logistics [142] and network management [143].

Considering that RTO systems which feature an architecture like the proposed in section 4.2 are not developed as monolithic programs, having also a distributed character, led to the idea of proposing a MAS as a novel internal organization for RTO systems. This way, more than one agent will be developed in order to implement architectural components.

The FIPA standards

Taking into account the application of RTO systems in the industry, one of the questions that came up was if there were already some standards defining how agents should be used in real applications. That is where the Foundation for Intelligent Physical Agents (FIPA) entered the scene.

FIPA [144] is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. Its specifications contain a collection of standards, which are intended to promote the inter-operation of heterogeneous agents and the services that they can represent. The standards address concerns related to agent communication, management and architecture.

In relation to agents management the standards present a normative framework within which FIPA-compliant agents can exist, operate and be managed. This way the logical reference model for the creation, registration, location, communication, migration and operation of agents is established.

An abstract agent architecture was also created and standardized in order to circumvent the impact of core specifications. Key aspects of the most critical mechanisms were abstracted into a unified specification. The overall goal of the approach was to allow the creation of systems that seamlessly integrate within their specific computing environment while inter-operating with agent systems residing in other environments.

Agents Communication

Communication is a topic of central importance in concurrent systems fields like MAS. Many formalisms and abstractions have been developed approaching this topic [145][146].

FIPA standards define that agents communication is accomplished asynchronously and by means of messages interchange. In a MAS scenario, an agent i cannot force agent j to perform some action a , nor to write data onto its internal state. This is because of the autonomous behavior that is tied to the concept of agent. An agent has full control over its internal state and its behavior. Besides, performing the action a may not be in the set of interests of agent j . Nevertheless, this does not mean that they cannot communicate.

What agent i can do is perform actions - communicative actions - pursuing to influence j appropriately. This is where speech act theories comes into the stage. The central ideal of these theories is that communication can and should be treated as an action. Speech actions are performed by agents just-like other actions, trying to bring their intentions to reality.

Several languages have born specifically for agent communications because of influence of speech act theories [5]. KQML [147] is such a message-based language. KQML defines a common format for messages. Each message has a *performative*, and a number of attribute/value pairs acting as parameters. Several versions of KQML with different collections of *performatives* in each were proposed during the 1990s [5].

ACL (Agent Communication Language) is a standard language for agent communications proposed by FIPA that is grounded in speech act theory as well and superseded KQML. A set of interaction protocols, each consisting of a sequence of communicative acts to coordinate multi-message actions, is also defined. The possibility to define domain specific ontologies, in order to provide content semantic for agents communication is also provided.

Agents cooperation and coordination

Negotiation has long been recognized as a process of some importance for MAS research. Generally, it aims to modify the local plans of agent (either cooperative or self-interested) in order to avoid negative (i.e. harmful) interactions and to emphasize the situations where potential positive (i.e. helpful) interactions are possible. Many researchers have proposed different negotiation mechanisms. Generally such approaches are based on operations research techniques.

A MAS can be constituted by agents that have not been designed and implemented by one single individual or several individuals with same objectives in mind. Therefore, agents might not share common goals. Based on the concept of autonomy, an agent must be prepared to make decisions about what to do at run-time, instead of having those decisions

hardwired at design time. In that decision making process, the presence of other agents in the environment brings to scene the need of been capable of dynamically coordinating its activities and cooperating with the others.

In a MAS where agents have different goals they must act strategically in order to achieve the stair of affairs they consider preferable. The encounters that will occur between them in that scenario resemble games. That is why the field of research known as *games theory* finds applications in MASs.

Organizations of Agents

Unlike heterogeneous individual agents that meet casually and interact trying to achieve its own independent objectives, a MAS can be strategically conceived with a global shared objective in mind. Even using this approach where an a priori intention exists from actors or designers aiming at a global common purpose, a lack of some organization specification in a MAS may lead to the lose of the system global congruence, caused by the agents autonomy.

An organizational specification is profitable for the MAS because it improves the efficiency of the system constraining agents behaviors and orienting them towards those that are socially intended [148] [149]. The improvement is achieved by means of establishing coordination and cooperation patterns as well as rules derived from the organizations theory. The field called *Organizations in MAS* collect researches about this topic having the concept of *organization* as a central starting point.

Fox et al. [150] consider an *organization* to be a set of constraints on the activities performed by a set of collaborating agents. This concept follows the one that views the process of bureaucratization as a shift from management based on self-interest and personalities to one based on rules and procedures [151]. According to Malone [152] an *organization* is a decision and communication schema which is applied to a set of actors that together fulfill a set of tasks in order to satisfy goals while guarantying a global coherent state.

When considering an *organization*, there are some concepts that are essential and inherent to it according to [149]:

- Division of activity types: Following the idea of “division of labor” for human organizations, this concept set that activities in *organizations* are differentially and not uniformly or randomly distributed across an organization. A taxonomy of activities inside the MAS can be defined according to its types.

- Integration: Interdependencies can be identified between different regions of activity inside an *organization*. In some sense, an *organization* is a set of constraints on the set of relationships between *organization*'s activity spaces.
- Compositionality/Recursivity: By means of aggregative and stabilizing mechanisms an *organization* can be composed of multiple overlapping structures and/or sub-organizations.
- Stability/Flexibility: Patterns of activity in an *organization* have stable features, as they are patterns, but they have also flexible features. From an architectural point of view, *organizations* can be considered an architecture. It defines stable constraining structures within which a certain degree of flexible activity is possible.
- Coordination: In order to be efficient, an *organization* needs to exhibit high coordination of its activities.
- Supra-Individuality and Roles: A *role* normally describes a consensually defined bundle of activity types that are prototypical types of behavior. *Roles* are supra-individual constructs because they map to activity types, not to concrete, specific individuals, which can be replaced.
- Rules: *Organizations* are rule-governed structures [153] [154].

The intention to link formal or human *organizations* theory to MAS models exists since several years ago. Study bodies such as Mathematics, Computer science, Mechanics, Thermodynamics, Sociology, Social Psychology and Ethology served as inspiration sources [155].

From a MAS perspective, an *organization* is a supra-agent pattern of cooperation of agents in the system. It is characterized by a division of tasks, a distribution of roles, authority and communication systems and contribution-retribution systems [156]. Phenomena and characteristics linked to the *organization* are considered supra-individual as they exist at a level independent of specific individual behaviors or attributes. All this results into a system that overpass its individuals' skills with an efficiency higher than the simple aggregation of efficiencies from its individual agents [149].

Patterns of cooperations inside MASs with organizational specification can be classified as *predefined* (defined by the MAS designer) or *emergent* (defined by the agents themselves at runtime) [157]. Elements like the organizational structure and its norms contribute to a predefined cooperation. On the other hand, organizational entity, institution, social relations and commitments lead to emergent or potential cooperations [155]. That motivated the two approaches that are classically used to describe or design an *organization*

for a MAS, having the concept of *organization* at the intersection of both. They differ basically in which is the first-class entity represented.

The first approach, called *agent centered*, takes the agents as the engine for the organization formation [157]. The functioning of the agents as *organization* is a result derived from the individuals behaviors, and there is no well defined specification of the global objectives. Actors or designers of such a MAS first focus on parts of the system-to-be (i.e. the agents). The approach sets that designing proper local behaviors and peer-to-peer interactions will lead to the satisfaction of the system global function as a result of complex interactions and dynamics within the agent society. In other words, the approach bets on the emergent cooperation in the organization. A drawback of this approach is that it often introduces unpredictability or uncheckability, since the global behavior is more than the juxtaposition of agents behaviors [158]. The point of view of emergent phenomena in complex systems for organizations is studied by the Self-Adaptive and Self-Organizing systems (SASO) community of the multi-agent domain.

The second approach, identified as *organization centered* [157], support the idea that the organization exists a priori (defined by the designer or by the agents themselves) and the agents ought to follow it [157]. Such a MAS, from the design phase, have a clear specification of global objectives, which is shared between the agents. Each agent must to observe the rules declared in that specification.

Contrary to the SASO approach, Coordination, Organization, Institutions and Norms in Agent Systems (COIN) community focuses on an organization-centered point of view, in which the designer defines the entire organization and coordination patterns on the one hand, and the agents local behaviors on the other hand. Agents may consider at runtime the constraints imposed by the organization as compulsory or possible guide-lines for the coordination of their local behavior. Using this approach, the actors or designer can ensure that the MAS will stick to the organization specification.

The notion of having software entities working with a common or global purpose, to optimize the chemical process, led to think in the system as an *organization* of agents. At the same time, the fact that the system was going to be centrally designed and brand new agent breeds were going to be created, an organization centered point of view with predefined cooperation was chosen for the MAS design. Nevertheless, the RTO system will not be a closed one. New agents developed by other entities can join the system and use its services, since they comply with the organization rules.

5.1.2 Object Oriented Design and Programming

With the introduction of computers in several applications during the 1970s, a demand for software development rose being difficult to be satisfied. The costs of software development and maintenance grew uncontrolled, giving rise to the so called software crisis.

To deal with this situation a change in the way to think about software programming was needed. In this new approach software specialist stopped to see programs as sequences of instructions and started to think about them as collections of interacting modules. New programming languages and paradigms appeared then, having as base simplicity and structure. One of the paradigms created during that time was the Object-Oriented Programming (OOP), having its root in the Simula programming language and its first implementation in the Smalltalk [159] programming language [160].

OOP can be defined as a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. In the kernel of the OOP is the concept of *object*. Computationally, an OOP is an abstract representation of a concept or entity of the real world. An *object* has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable [161].

OOP involves several key concepts that make easier the development and maintenance of complex software systems. The first of them is the concept of *abstraction*. An *abstraction* denotes the essential characteristics of an object that differentiate it from all other kinds of objects and therefore provide well-defined conceptual boundaries. An *abstraction* focuses on the outside view of an object, and so serves to separate an object's essential behavior from its implementation. Deciding upon the right set of abstractions for a given domain is the central problem in object-oriented design. Using abstractions is a natural way to deal with complexity.

Another main concept is *encapsulation*, which define the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior. *Encapsulation* serves to separate the contractual interface of an abstraction and its implementation. Both, *abstraction* and *encapsulation*, can be considered as complementary concepts: *abstraction* focuses upon the observable behavior of an object, whereas *encapsulation* focuses upon the implementation that gives rise to this behavior [161].

A *class* is a template that resumes the structure and behavior that are common to a set

of objects. Every *object* is an instance of a *class*. Relationships are established between classes. One type of these relationships, called *inheritance*, is central to OOP. *Inheritance* is the mechanism by means of which a *class* inherits the structure and behavior of a superclass this way allowing the code reutilization.

When a *class* inherits from a superclass, the inherited behavior can be redefined. This allows the child *class* to act in a different way than its ancestor in some specific situations. This is the base of another central OOP concept known as *polymorphism*, which is perhaps the most powerful feature of OOP languages next to their support for abstraction, and it is what distinguishes OOP from more traditional programming.

An *interface* is a concept that models objects' behavior beyond the *class* concept. An *interface* defines the possible operations or services that are available for some breed of objects, without defining how those operations are implemented at all. In that sense, an *interface* defines a protocol, a convention, on how requesters could interact with an object. In order to let clear the difference between a class and an interface, it could be said that *classes*, even been abstract, could contain internal structure or behavior for the objects that will inherit from it. By other side, and interface has nothing to do with data, internal structure or behavior, they just define the *what*, not the *how*. As summary, *classes* implement *interfaces*.

The decision to use object-oriented design and programming during the implementation of the RTO system was made after valuing the tools those paradigms offer to adapt a real problem to a computational model and the vantages they provide in terms of modifiability and extensibility.

5.2 Technologies

5.2.1 EMSO

One of the constraints defined in the requirements and accepted during the development process was the use of the tool Environment for Modeling, Simulation and Optimization (EMSO). EMSO is an equation based environment which provides a language for dynamic or steady-state processes modeling and an open source library of models, all this under the name EMSO Modeling Library (EML).

An EML model consists of a mathematical abstraction of some real equipment, process, piece or even software. Examples of models are the definition of a tank, pipe or a PID controller. Each model can have parameters, variables, equations, initial conditions,

boundary conditions and submodels, that can include submodels themselves. Models can be based in pre-existing ones, and extra-functionality (new parameters, variables, equations, etc.), so composition (hierarchical modeling) and inheritance are supported.

Every parameter and variable in a model is based in a predefined type and have a set of properties like a brief description, lower and upper bounds, unit of measurement among others. As models, types can have subtypes and the object-oriented paradigm is implemented.

EMSO is capable to solve high-index differential-algebraic system of equations and has a symbolic and automatic differentiation systems. Implementation of nonlinear-algebraic solvers is also possible. Functionalities can be extended thanks to a plug-ins mechanism.

5.2.2 JAVA

Java is a programming language originally developed by Sun Microsystems, released in 1995 and built exclusively as an object-oriented language. Much of its syntax derives from C [162] and C++ [163] languages but has a simpler object model and fewer low-level facilities. In Java, almost everything is an object and therefore all code is written inside a class.

Properties of Java, listed next, motivated its selection as the language for general implementations during the RTO system development:

- It is simple, object-oriented, and familiar.
- It is architecture-neutral and portable.
- It is robust and secure.
- It executes with high performance.
- It is platform independent.
- It is interpreted, threaded and dynamic.
- It contains built-in support for using computer networks.
- It was designed to execute code from remote sources securely.

Platform independent, means that programs written in Java run similarly on any supported hardware/operating-system platform, allowing the user to compile one program once and run it anywhere.

A Java installation typically includes its Software Development Kit (SDK) and Java Runtime Environment (JRE), which is a subset of the SDK, the primary distinction being that in the JRE, the compiler, utility programs, and other necessary files are not present.

5.2.3 JADE

The Java Agent Development Framework (JADE) [132] is an open source framework for peer-to-peer agent based applications fully implemented in the Java language. It provides a platform that fully implements the FIPA specifications and simplifies the development of multi-agent systems [128]. The platform includes a set of graphical tools for helping in the debugging and deployment development phases.

A multi-agent system built based on JADE, can be distributed across machines controlling the configuration using a remote user interface. Thanks to the use of the Java language, the machines can have different operating systems. When agents are created, they are automatically assigned a globally unique identifier and an address that are used in the communication process. Each agent runs as a separate process and will be capable of transparently communicate with one another.

The platform supplies a library of interaction protocols, which model typical patterns of communication. For this purpose application-independent Java classes skeletons, which can be customized with application-specific code, are available.

JADE provides efficient transport of asynchronous messages. It selects the best available means of communication between agents. Simple software libraries and graphical tools are also provided to both locally and remotely manage agent life cycles. The platform performs automatically the checking and content encoding of user defined domain specific ontologies. Programmers are able to select preferred content languages and ontologies. New content languages to fulfill specific application requirements can also be implemented.

5.2.4 JaCaMo

Having decided to use an organization of software agents as the approach to build the RTO system, and JADE and Java as underlying layers, it made sense to look for a technology that could integrate all concepts. In principle, the support of some agent architecture—i.e., BDI—was a bonus requirement in order to do not need to implement one from scratch.

The reading of a paper authored by Bossier et al. [164], published as result of a collaboration work involving four universities, made us meet JaCaMo, a software framework for

building multi-agent systems.

JaCaMo integrates semantically three technologies (Jason, CarTaGO [165] and MOISE [166]), each of them representing a different programming dimension (Agent, Environment and Organization) respectively. The intention was to provide a uniform programming model making easier to combine these dimensions.

Considering the benefits it provides in terms of high level agents development and technology integration, JaCaMo framework was chosen for implementing a RTO prototype system. Next sections describe each technology separately, in order to get a better understanding.

Jason

Jason [126] is a platform that provides a high level AOP language for the development of multi-agent systems. It allows programmers to focus on implementing how the “mind” of agents works and the reasonings that can be used to reach its goals. It is a variant of AgentSpeak, so it implements the BDI model. Among other things, Jason enrich AgentSpeak with an event mechanism allowing agents to react to their environment perceptions.

CARTaGO

CARTaGO (Common ARTifact infrastructure for AGents Open environments) [165] is a general purpose framework and infrastructure for programming virtual environments used in multi-agent systems. It is based on the Agents Artifacts (AA) meta-model [167] for modeling and designing multi-agent systems, which proposes that agents interact with their environment making use of artifacts.

The main idea behind CARTaGO is that the environment can be used as a first-class abstraction for designing a MAS. It can be considered a computational layer that encapsulates functionalities and services that agents can explore and consume at runtime [168].

In CARTaGO, developer find a simple programming mode to create software environments executed as a dynamic set of computational entities of different kinds called artifacts, which are collected into open workspaces. Inside a network, workspaces can be distributed among various nodes. Agents of different platforms can join the workspaces so as to work together inside such environments.

CARTaGO is not bound to any specific agent model or platform: it is meant to be orthogonal with respect to the specific agent model or platform adopted to define agent

architecture and behavior. Nevertheless, CArtAgO is especially useful and effective when integrated with agent programming languages based on a strong notion of agency, in particular those based on the BDI architecture.

Moise

Moise [169] is a software framework that implements a programming model covering the organizational dimension. It includes an organization modeling language and management infrastructure, and support for organization-based reasoning mechanisms at the agent level.

6 FIRST PROTOTYPE. APPLICATION TO A CASE STUDY

This chapter describes a first RTO system prototype that was developed applying concepts from the MAS field and using just the JADE platform. The implementation made part of the results from the collaboration project between USP and PETROBRAS. At the time the prototype was developed, the design of the software architecture proposed in chapter four for RTO systems had not finished nor was formalized. That is why, several concepts and designs were refined in a second iteration, when the SA was ready, and will be presented in the next chapter.

The idea behind applying the software agent paradigm to the RTO domain is to propose and build a community of agents whose central goals are the support of the main RTO tasks. With that in mind a set of interactive agents, each of them with a very well defined set of services and responsibilities, was conceived and implemented. How EMSO was adapted to join the general structure is described as well. The prototype was applied to the Williams-Otto case study [170].

6.1 RTO ontology

As with the use of the software agent paradigm the RTO cycle is implemented by means of interaction of agents, the information generated at each step needs to be represented with a well-defined syntax inside the interchanged messages. According to FIPA terminology, that syntax is known as a *content language*. Parsing the content of a message the receiver agent can extract each specific piece of information, but it must have some shared understanding with the sender about the concepts and actions described with the content language. The set of concepts and the symbols used to express them are known as an *ontology*. Ontologies are typically specific to a given domain and has to do with determining what classes of entities are needed for a complete description and explanation of the objects, properties, events, processes and relations present in that domain [171].

The support for content languages and ontologies provided by JADE allowed the definition of a RTO ontology to be used for messages interpretation. In JADE, an ontology contains predicates, terms, actions and aggregates. Predicates are expressions that say something about the status of the world and can be either true or false. Terms are expressions identifying entities (abstract or concrete) that exist in the world and that agents may reason about. They are further classified into Primitives (atomic entities such as strings and

integers) and Concepts (entities with a complex structure that can be defined in terms of slots). Actions are special concepts that indicate actions that can be performed by some agents. Aggregates are entities that are groups of other entities.

Examples of concepts included in the RTO ontology described in JADE are *Process Variable*, *Process Tag*, *Objective Function Value*. Examples of predicates include: *Process is at steady-state* and *Optimization not solved*. Actions include: *Get last process sample*, *Solve Optimization problem* and *Update set-points*.

6.2 Covering the RTO ontology

From the beginning, it was identified the need to design and implement a software framework which would serve as a basis for the subsequent development of RTO concerns.

A software framework is an abstraction in which software providing generic functionality can be selectively overridden or extended, thus providing a shell to develop specific domain applications. It furnishes building blocks for a family of systems and places where adaptations should be made, known as extension points [119]. A default behavior guarantees the framework achieves its mission in a predefined way. In an object-oriented environment, the extension points consist of abstract and concrete classes and interfaces which behavior can be modified adding new code or overriding existing.

The framework described next, referred as **RTOF**, can be considered a direct contribution of this work. It covers several concepts and concerns involved in the implementation of RTO systems. The abstractions and concepts were modeled as interfaces and classes using object-oriented analysis and design [161] and implemented afterward in the Java programming language.

The central usability of **RTOF** resides in the fact that, as a framework, it can be extended by means of two principal actions: adding new functionalities to expand its scope or modifying actual features to handle different approaches and technologies related to RTO concerns. First action can be executed just inheriting from **RTOF** classes and interfaces and adding functionalities to the new entities. Second action can be done inheriting and overriding actual functionalities, just to do the same tasks in a different way. That is the power of object-oriented programming applied to an specific domain, in this case, RTO.

6.2.1 Framework description

Every class or interface modeled in **RTOF** is derived from some element listed in the concepts map showed in figure 6.1. Cardinalities of both sets (classes/interfaces in **RTOF** and concepts from the map) are not the same. There are more classes/interfaces in the framework that concepts represented in the map. Two groups of entities can be identified inside **RTOF**: those modeling general abstract concepts and those that model plant process related elements.

Figure 6.1 – Map of concepts covered by RTOF

One of the main purposes of **RTOF** is to serve as a bridge between the plant and the RTO system. Reading data from plant information systems and communicating with control interfaces to ask for set-points update are two main concerns covered in **RTOF**. With that in mind, the framework was designed with generality, been possible to override its classes and interfaces when needed, in order to handle information technologies that could be found in a real facility.

Concerns that appear in a typical RTO implementation, like data treatment, gross errors and steady-state detection are also covered in **RTOF**. The framework design provides spots where these RTO steps can be implemented in diverse configurations. Next sections describe main classes and interfaces implemented in **RTOF**.

6.2.2 Java Classes and Interfaces

The Java platform organizes code into hierarchical packages. A package is a logical aggregation of related code files and packages. **RTOF** packages are listed below. A general description is presented for each of them and their classes/interfaces are also listed and described. Elements that can be considered as natural extension points—i. e., those that are more probable to be extended and/or overridden seeking extensibility— appear underlined.

Package: `br.usp.pqi`

This is the top level package in **RTOF**. It includes all the entities within the framework. All other packages are branches from this central trunk.

Variable: Class representing the abstract mathematical concept of a domain variable.

VariableInstance: Class representing an instance or observation of a domain variable.

FloatVariable: Class representing a domain variable that takes float values as domain.

FloatVariableInstance: Class representing an instance or observation of a float domain variable.

ITimeDimensionable: Interface for an abstract entity to which an instant of time with precision of milliseconds can be attributed.

DatePeriod: Class for objects that represent a date period, with an initial and a final date.

IDataSource: Abstract interface for objects than can be considered as generic source of data.

ObjectDeposit: Class representing an object that can act as a generic vertical container where objects can be stacked.

Package: `br.usp.pqi.event`

This package contains classes and interfaces that model some concepts related to computational events. An event is defined as an occurrence of some importance that took place as part of the running of a software program.

SystemEvent: Abstract class representing a generic event. Every system event is an derivation of this class and carriages another object with specific information about the event.

IEventSource: Interface for an object that can be considered the source of events.

EventHandler: Interface for an object that can handle triggered events.

StreamLineRead: Class representing an specific event object fired when a stream line has been read.

Package: br.usp.pqi.process

This package contains entities that model concepts involved in the communication with the plant information system, mainly those related to the extraction of data about the process state.

ProcessTag: Class representing a variable in the plant process flowsheet. It is normally uniquely identified by the plant information/control system.

TagDescriptor: Class representing an object that contains meta-data (identifier, label, description, variable name, etc) about an specific **ProcessTag**.

ITagsCatalog: Interface for an object that can be considered a catalog where information about **ProcessTag** objects can be looked up and instances of **TagDescriptor** class can be obtained.

TagsCatalog: Class providing a default implementation of the **ITagsCatalog** interface.

ITagsDataSource: Interface for an object that can be asked to retrieve the values of **ProcessTag** referenced by their identities.

TagsDataSourceConfig: Class representing an object that contains details about accessing or connecting to an object with the **ITagsDataSource** interface.

DBTagsDataSource: Class containing an abstract implementation of the **ITagsDataSource** interface for using physical databases as source of **ProcessTag** values.

PSQLTagsDataSource: Class containing an specific implementation of **DBTagsDataSource** class using PostgreSQL [172] as the underneath database technology.

TagMeasure: Class representing a measure of a **ProcessTag** taken at some point. It contains the value of the variable represented by that **ProcessTag** at that time.

ProcessSample: Class representing sample taken from the plant information system. It was taken at some point and contains several **TagMeasure** instances.

SamplesDeposit: Class representing an object that acts as container for **ProcessSample** instances.

SamplesPeriod: Class which instances represent objects that contains several plant samples (**ProcessSample** instances), taken over a period of time.

ProcessDataTask: Abstract class representing some computational task related with the plant process.

Package: `br.usp.pqi.process.sampling`

This is a sub-package containing entities for modeling the process sampling actions.

TagsSamplingTask: Class containing a default implementation of a computing process dedicated to sample the plant retrieving data about the actual variable values. Objects from this class use some implementation of the **ITagsDataSource** interface to read plant states and put them into a **SamplesDeposit** object. This is another natural extension point for the framework.

SamplingConfig: Class that functions as a data structure containing default parameters for a plant sampling task (**TagsSamplingTask**) run. Start delay, execution frequency, and time window size are some of the parameters. The behavior of plant sampling actions can be extended inheriting and overriding this class.

IProfitFormula: A formula can be defined in the framework using an implementation of this interface to calculate, at real time, a value that can be considered as real-time profit for the plant. The idea is to use this value as a real-time feedback for the last set-points change calculated by the RTO program and applied by the plant control system.

Package: `br.usp.pqi.process.sampling.event`

This package collects the system events raised during plant sampling task runs.

TagsSampledEvent: Class representing an instance of a specific event that is raised every time the plant is sampled and new values for the sampled variables was obtained.

ProcessSampleCreatedEvent: Class representing an instance of a specific event that is raised every time an instance of a **ProcessSample** object is created for handling internally in the framework the new plant variable values.

ProfitCalculatedEvent: Class representing an instance of a specific event that is raised every time the real-time profit of the plant is calculated using some implementation of the **IProfitFormula** interface.

Package: `br.usp.pqi.process.analysis`

This package gather elements related with making analysis over the plant data.

TagsAnalyzer: Class for objects that sample the plant through some implementation of the `ITagsDatasource` interface. **TagsAnalyzer** objects perform analysis of read data based on a dynamic list of tasks that can be assigned to it. Each task is implemented with a purpose in mind. A default implementation for a task that performs steady-state detection is provided.

Package: `br.usp.pqi.process.analysis.ss`

This package, a brand from the analysis topic, is focused in the steady-state detection concern.

ISSDetectionMethod: Interface that models the behavior of an object that can be considered a steady-state detection method.

SSDetectionMethod: Abstract class that contains a provided default implementation of the `ISSDetectionMethod` interface. The framework can be extended with new steady-state detection methods inheriting and overriding this class.

CaoRhinehart: Class containing a default implementation of the Cao-Rinehart steady-state detection method described in [46].

SSDetectionMethods: Utility enumerative type for referencing, parsing and instancing the steady-state detection methods implemented in the framework.

SSDetectionTask: Class containing a default implementation for performing steady-state detection tasks.

SSDetectionConfig: Class containing data structure for storing parameters for a steady-state detection task (**SSDetectionTask**) run. Start delay, execution frequency, and detection method are between the parameters. The behavior of steady-state detection tasks can be extended inheriting and overriding this class.

Package: `br.usp.pqi.process.analysis.ss.event`

This package collects the system events related to the steady-state detection task.

SSAnalyzedEvent: Class representing an instance of a specific event that is triggered every time a steady-state detection analysis has taken place.

SSAnalyzedEventInfo: Class representing an object that contains specific information about a triggered event and that is carried inside the event object itself.

Package: br.usp.pqi.process.optimization

This package covers concepts related with the process of managing candidate solutions for the plant operational point optimization.

ProcessOptimization: Class representing a candidate solution for the RTO problem of finding a better operational point for the process. An object from this class contains a set of values for the process set-points, a timestamp indicating when that solution was found, a reference to the solution source, and the value of expected profit if the suggested set-point values are applied.

IFeasibilityChecker: Interface that must be implemented by objects that can act as checkers of the feasibility of candidate **ProcessOptimization** instances. All the feasibility checkers can be considered filters between the RTO system and the plant control system, avoiding to pass set-points with values that do not make sense considering the actual plant conditions and physical limitations.

FeasibilityCheckerDescriptor: Class that storage metadata like *name*, *description* and implementing *class name*, for an specific feasibility checker implementation.

FeasibilityChecker: Abstract class that provides a default implementation for the methods declared in the interface **IFeasibilityChecker**. All classes intended to act as feasibility checkers should inherit from this one and add or override behavior, extending the framework.

SetPointDistanceChecker: Framework class that contains an specific implementation of the **FeasibilityChecker** class. The checker implemented here determines if the candidate solution set-point values are too much distant from the actual operational state.

IPerformanceCounter: Interface for an entity that acts as a container for statistics related to the performance of an object.

Package: br.usp.pqi.run

This general package serves as container for classes and interfaces that model concepts related with computational process runs.

ISystemTask: Generic interface for objects that represent any system task that can be run as a computational process.

ILoopingTask: Generic interface for objects that represent any system task that run in loop— i. e., that keep running until desired.

LoopingTask: Abstract class containing a default implementation of the **ILoopingTask** interface. Any computational process related to RTO can be included in the framework inheriting from this class and overriding it.

RunParams: Abstract class representing the set of parameters that a system task expect for running.

IExternalToolDriver: Interface for an object that will act as driver or interface with a computational process that is external to the framework or RTO system.

Package: br.usp.pqi.run.emso

This package was designed specifically for the integration of EMSO with the framework.

IEMSOCDriver: Interface that overrides the **IExternalToolDriver** interface, in order to contain all operations that are available on the EMSOC driver, which according to the framework design, is the object that handles the EMSOC computational process. The operation **getResults()**, which reads and returns the results of optimization or parameter estimation operations, is present in the interface declaration.

EMSOCDriver: Class containing a default implementation of the **IEMSOCDriver** interface.

SolveFSParams: Class representing a data structure containing the default parameters needed to be past in order to solve a problem in EMSOC (Examples of those parameters are: The path to the flowsheet file, the problem name and the results file name).

Package: br.usp.pqi.util

A package containing utilitarian classes.

DatePatterns: Provides the most common used date formats, in order to help the developer when forming text containing dates.

InputStreamProcessor: Abstract class representing an entity that processes an input stream running in a separated thread. A derivation of this class was used in the development of the EMSOC tool.

RegExp: Class used for parsing regular expresions.

RLTReader: Objects from this class can be used to read data from files of RTL format, which is a format specific to EMSO used to store run results.

StdOutput: Utility class wrapping the system standard output. Used whenever the developer needs to output some text.

TextFile: Class used to instance objects that can store data to a text file.

6.3 A MAS in JADE for RTO

In this system prototype, the RTO cycle is accomplished by the iterative interaction of the agents. GUIs were developed for each agent, and all interchanged messages are visualized in each agent's Graphical User Interface (GUI). The classes, structures and features resulting from the object-oriented design enable the agents to make easier for users the development and deployment of an RTO solution in a practical application. New instances can be designed and incorporated to the RTO community of agents in order to improve or gift the system of a new feature. Each agent is configurable by means of its GUI or an XML [173] file.

DIRECTORY FACILITATOR agent

The FIPA specifications declare the necessity of a special agent whose mission is to serve as a kind of “yellow pages” service. This service allows agents to publish descriptions of one or more services they provide allowing other agents to easily discover and exploit them. Any agent can both register (publish) services and search for (discover) services. Registrations, de-registrations, modifications and searches can be performed at any time during an agents lifetime.

The JADE platform provides a special and ready to use agent named DIRECTORY FACILITATOR (DF) to fulfill the yellow pages services requirement of the FIPA standards. One of the first tasks accomplished by agents once they are instantiated in the platform is to declare its services. An agent wishing to register one or more services must provide the DF with a description that includes its own identification, a list of provided services and optionally the list of languages and ontologies that other agents must use to interact with it. Each published service description must include the service type, the service name, the languages and ontologies required to use the service and a collection of service-specific properties in the form of key-value pairs. An agent must provide the DF with a template description when it wishes to search for services it. The result of the search is a list of all the descriptions that match the provided template [132].

PROCESS SENSOR agent

PROCESS SENSOR is a custom agent proposed in the system's design. This agent, by means of its structures and behavior, offers to the user the possibility to deal with key RTO concerns like data ingestion and treatment, gross error detection and steady-state

detection as well.

The agent serves as an interface between the real plant and the RTO system. That is, the SENSOR has the mission to feed the RTO system taking data at real time from data collection tools running in the plant information system. One example of that interaction is the communication of the agent with an instance of the Plant Information (PI) system that could be installed and running in the plant. The intention is that any data needed inside the RTO system and provided by the plant information system can be asked to the PROCESS SENSOR. The object-oriented design used in the internal organization of the agent, allows the implementation of custom plant-agent communication modules.

As the PROCESS SENSOR reads data from the plant it is capable to apply statistic filters to discard gross data errors. Several chained filters can be applied. Using the object-oriented fundamentals, the user can incorporate new filters to the process.

Another important feature of the PROCESS SENSOR is its capability to analyze the process variables in order to detect if the process is at steady-state. If it is the case that the process is stable, the agent provides a representative value for all the analyzed variables. The agent design, allows the user to switch between several steady-state detection methods and to implement new ones. As the most part of steady-state detection methods uses a data window to estimate the process stability, the PROCESS SENSOR can stay with the last read information as a data window. To deal with several methods, the agent exposes some configurable parameters, like the sampling frequency and data window size, which can be tuned by users. At the same time, each detection method can have its specific parameters declared in the configuration file.

The PROCESS SENSOR is an agent that is always waiting for a petition of data or information about the actual state of the process. It also offers a GUI showing the actual state of measured process variables, the result of each steady-state detection iteration and values for the variables from the last steady-state occurred. Agent's parameters and behavior can also be modified using the GUI.

OPTIMIZER agent

OPTIMIZER is a custom agent that is crucial for the RTO environment. Its main goal is to solve optimization problems specified in some modeling language. The agent is always waiting for a request about some optimization problem to solve. The request message should contain the details of the problem. Its response to received messages will include the optimization solution. The incoming optimization request and the solution values are

visualized in the agent's GUI.

As an optimization problem needs to be specified using some modeling language, the agent have to be capable to read and understand the defined problem. Taking that in mind, the internal architecture of OPTIMIZER was designed including an abstract implementation of a general optimizer and specific implementations for performing concrete optimizations according to a particular modeling language.

The first specific implementation provided used the EMSO simulator engine and its modeling language (EML) as base for solving optimizations. A careful and special work needed to be done in order to accept the constraint about the use of EMSO and to integrate it with the agents platform as the simulation and optimization engine. After considering some options it was decided to extract the EMSO kernel, letting out all the graphic user interface (GUI) functionalities, and turn it into a command line tool. That work was done building abstract software structures using the C++ language and reusing the EMSO services from them. The resulting software was called *EMSOC*. Whenever the OPTIMIZER agent is instantiated, it starts the EMSOC module, containing all necessary elements to solve problems defined in the EML language.

ORCHESTRATOR Agent

Unlike other agents that stay waiting for requests, ORCHESTRATOR was conceived as an active component that starts conversations with other agents. The RTO cycle needs a starter. That is the first role of ORCHESTRATOR. That software unit, by means of its behavior, defines also the RTO approach that will be followed in the system. The agent was designed internally to makes easy the implementation of several approaches. By means of its GUI, the user selects the RTO approach to apply. From that point, the actions taken by the agent respond to the script defined in the internal approach implementation.

PROCESS ACTUATOR Agent

PROCESS ACTUATOR serves as a bridge between the RTO system and the plant control system. Its main mission is to stay communicated with the control system waiting for a request to update the process set-points. Once a request is received, containing the optimal operational values the Actuator tries to send an order to the control system to lead the plant to that point.

As with the Optimizer agent, an abstract behavior of the Actuator was defined and implemented. Its internal design allows the user to implement several variants of the

communication with the control system of the plant. If that communication will be with an advanced control module like MPC or directly with the regulatory control depends of each case.

With PROCESS ACTUATOR and PROCESS SENSOR agents, the logical boundary between the plant and the RTO system was well delimited.

6.4 The MAS in action

An implementation for the classical RTO approach or MPA was provided. In this approach, the RTO cycle starts with a request from the ORCHESTRATOR to the DIRECTORY FACILITATOR looking for some agents providing information about the process state. If the PROCESS SENSOR is running at that moment, the DIRECTORY FACILITATOR returns the PROCESS SENSOR's address and from there the ORCHESTRATOR can communicate with it directly. The ORCHESTRATOR sends then a request asking for the process state. If the response says that the process is at steady-state, the ORCHESTRATOR can continue with the next step, parameter estimation.

As parameters estimation task is after all an optimization problem, ORCHESTRATOR will look for some agent offering the Optimization service. If the PROCESS OPTIMIZER is present, a request is sent to it asking for solving the problem. The request will also include the representative steady-state variable values retrieved from the PROCESS SENSOR. Once the model parameters are recalculated, the final optimization is performed using the same mechanism: asking for the optimization service and sending a request.

Once the last optimization problem has been solved and the values of the optimal operational point have been determined, ORCHESTRATOR asks to the DIRECTORY FACILITATOR for an agent providing plant set-points update service. If some agent is alive and offering that service, ORCHESTRATOR sends a message containing the optimal operational values to be applied by the plant control system. The described interaction of agents running the RTO cycle is presented in the next figure.

6.5 Application to a case study

The MAS prototype was configured to apply RTO the Williams-Otto case study [170] using a MPA approach. This case study is a well-known process that occurs inside a Continuous-Stirred Tank Reactor (CSTR) and has been used for the development and comparison of RTO strategies by several authors.

Figure 6.2 – RTO cycle implemented with the MAS.

The reactor is fed with pure streams of components A and B , represented by F_A and F_B respectively. These components react producing an intermediate component C that reacts with B to produce the desired products P and E . A side reaction happens also between components C and P , producing the product G that has no commercial value and is waste. Next figure shows the reactor and the equations of reactions.

Figure 6.3 – Williams-Otto reactor and equations.

The process was modeled at steady-state and based in the mass balances using the reactor Temperature (T_R) and flow rate of B (F_B) as controlled variables. The flow rate of reactant A (F_A) and the mass holdup W were kept at 1.8275 kg/s and 2105 kg respectively.

The economic objective was to maximize the profit given by:

$$\phi = 1143.38 X_P F_R + 25.92 X_E F_R - 76.23 F_A - 114.34 F_B$$

where X_P and X_E are the mass fraction of P and E in the reactor outlet stream (F_R). In the implementation of a RTO prototype for this case study the described agents were

used (PROCESS SENSOR, OPTIMIZER, PROCESS ACTUATOR).

To imitate a real RTO cycle, a dynamic model of the reactor process and a simulation module were developed in MATLAB [174] by the project team. The simulation module takes set-point values from a text file and stores the model outputs into another file. Using the Pentaho Data Integration (PDI) tool [175] the model outputs were updated constantly inside a PostgreSQL [176] database.

A custom module for the communication of PROCESS SENSOR with the database was developed as well as a custom module for the PROCESS ACTUATOR set-points storing. Next figure resumes the described configuration.

Figure 6.4 – Environment created for the Williams-Otto RTO prototype

For steady-state identification, the method proposed by Cao and Rhinehart was implemented and incorporated to the PROCESS SENSOR agent. The solver used for optimization was the Ipopt algorithm [177].

Figure 6.5 shows the agents' GUIs of the running prototype. All the interfaces offer to the user two tabs identified as *Logs* and *Messages*. The first of them, provides a log of all the activities and steps that are been carried out by the agent. That way the user can trace the process. The second tab collects all the messages sent and received by the agent during the run.

The top left corner of the picture shows the PROCESS SENSOR agent's interface. The first tab, identified as *Process* is dedicated to show results from the two main activities

of the agent: the process sampling and analysis. At the left part there is data about the last sampling time, and values collected from the process tags at that moment. The central part shows two timeline charts showing values for selected tags and the process state as a whole. To include tags in the upper graphic the user can select it from the *Tags Values* table and press the *Chart* button. The right section of the window shows the last time the process was analyzed and the result from that analysis. In case the process is at steady-state, values for the stable variables are showed as well.

Figure 6.5 – The RTO prototype running the Williams-Otto case study.

In the *Settings* tab specific configurations can be set for the agent. That tab is where the user can change parameters such as the sampling and analysis frequency, the tags to be sampled and considered in the analysis, and the steady-state detection method.

The window at the right corner corresponds to the OPTIMIZER agent. When the parameter estimation and optimization steps of RTO are in execution, output from the process can be seen in the *Optimization* tab. Location of the parameters estimation and optimization flowsheet, model parameters, free optimization variables, between other, are configuration elements defined in the *Settings* tab.

The window identified as *RTO Orchestrator* corresponds to the interface of ORCHESTRATOR agent. In the first tab, the actual phase of the RTO cycle can be verified, been possible to start/stop it at any time. The result from last iteration is displayed at the bottom. The RTO approach is defined in the *Settings* tab.

Details of the steady-state model, flowsheet and optimization files in EML can be found in appendixes A, B and C.

7 SECOND PROTOTYPE. APPLICATION TO A REAL CASE

This chapter describes the implementation of another RTO prototype approaching the implementation of RTO on an industrial-scale depropanizer column from Paulínea refinery (REPLAN), owned by Petrobras S.A., where a Vapor Recompression Distillation (VRD) process is carried out.

VRD is a well-known highly integrated energy process which is widely used to split close-boiling mixture, such as propylene and propane. One of VRD's main characteristics is that additional mechanical energy is added to the overhead vapor stream by a compressor. That stream is then used to boil up the mixture in a reboiler, reducing this way the total amount of demanded energy.

An schematic structure of the VRD process is showed in figure 7.1. First, a low molecular weight hydrocarbon mixture (mainly propylene and propane) enters the distillation column, where high-purity propylene (99.95%) is obtained as overhead product stream D , and propane (95%) is obtained as the main product at the bottom stream B . The overhead stream is mixed with vapor stream from the distillate drum, and then, it is compressed to increase its condensing temperature. After that, the largest part of the compressor outlet stream feeds the reboiler (F_{boil}), while the rest (about 10%) is condensed with cooling water (F_{cool}) to control the column pressure. Subsequently, the propylene streams (hot stream) from reboiler and condenser expand through throttle valves, returning to the distillate drum, where a portion of the liquid is sent to the column as reflux stream (R), and the other part is stored as high purity propylene (D) [178].

Following an equation oriented approach, a mathematical model of the unit was developed in EML by the project team [179]. Some sections from the model are extracted in appendix D. The VRD process features a highly interlinked structure and a large number of nonlinear equations when expanded (around 8000 in the REPLAN case).

This time, the implemented prototype attended the proposed architecture, followed again the MAS paradigm and used the JaCaMo platform. How architectural components turned out into functional pieces is presented. From now on, let's use the acronym **RTOMAS** to refer to the implemented prototype.

Figure 7.1 – Schematic representation of the VRD process.

7.1 Building the MAS

Having chose the technology to implement RTOMAS, it was time to take decisions about which of the available elements of JaCaMo should be used to develop each architectural component. In other words: which of the system functional requirements should be implemented as artifacts and which as agents?

Two criteria helped us to answer prior question. The first one is about the shift of system's reactive character, during the design process, to specific architectural components. The more reactive the component seems to be, the more it should be implemented as software agents. The other criteria talks about which system functionalities looks like agents actions over the environment. According to CArtaGO's philosophy, those actions should be implemented as artifacts.

As figure 4.2 suggests, since early design steps, reactive behavior got more localized in OPTIMIZER and its internal subcomponents. It is inside that component where a reaction is triggered in response to some sensing action. On the other hand, PLANT DRIVER component is the entity that groups all actions performed over the plant, so it seems right to implement it as environment artifacts.

7.1.1 Implementing PLANT DRIVER component

As figure 4.3 shows, responsibilities of PLANT DRIVER were assigned to two internal subcomponents: PROCESS INFORMATION SERVICE and PROCESS CONTROL INTERFACE. Next sections details how those subcomponents were implemented.

Implementing PROCESS INFORMATION SERVICE

In order to implement functionalities covered by PROCESS INFORMATION SERVICE, a specific artifacts breed was conceived, calling it *Tags Board*. It was designed after the metaphor of having a *board* which provides real-time information about process variables' value, called here as *Tags*, and process state.

Several instance of *Tags Board* can be used inside the RTO system, targeting several *Tags* or subprocesses. The idea behind *Tags Board* is to provide its services to whichever agent or artifact present in the RTO system, in order to help them to do the work.

Once a *Tags Board* is "on", it starts reading process variables from plant's control systems and brings it easy to be visualized by system's users. At the same time, the board analyzes the process state, determining if it is steady or transient. DATA INGESTOR and ANALYZER internal components, which communicate by means of SHARED REPOSITORY architectural connector, conform the internal architecture of every *Tags Board* artifact.

The steady-state detection method proposed by Cao and Rhinehart [46] and implemented as part of RTOF was used as the default method of *Tags Board* instances in the prototype implementation.

The features *observable properties* and *operations*, introduced by CArtAgO framework, allow *Tags Board* instances to provide data and actions to other components and system users. Properties and operations can be extended depending on what concern will be approached or which are the user's needs.

Some observable properties implemented in *Tags Board* are:

connected: Says if the board is connected to plant control system, enabled to read data from it.

sampling: Says if the board is sampling the plant.

last_sampling_time: Shows the last time the plant was sampled by the board.

process_state: Says if plant is at *steady-state* or *transient*.

last_analysis_time: Shows the last time the plant state was analyzed.

instant_profit: Shows the real-time profit of the process, based on some formula.

Operations include:

setupBoard: Used to setup the artifact, passing all the necessary parameters. Variables to be sampled, sampling frequency, and size of the time window for staging samples are examples. Other parameters include the steady-state detection method to be used and the state analysis frequency.

startSampling: Orders the artifact to start its process monitoring, sampling and analyzing the process state. This operation represents the concretization of agents intentions to sense the environment.

getSamples: Returns a `SamplesPeriod` instance containing samples with all Tags the *Tags Board* has access to. It expects a `DatePeriod` instance as parameter, been possible to request samples from some time ago.

stopSampling: Stop the monitoring activity of the artifact.

Whenever a new steady-state is identified, *Tags Board* emits a signal that can be heard by all agents that are focused on it. An object containing a representative steady-state is sent as part of the signal.

Along with `CARTAgO` classes, packages from RTOF were used in the Java implementation of *Tags Board* artifacts. Between them:

- `br.usp.pqi`
- `br.usp.pqi.process`
- `br.usp.pqi.process.sampling`
- `br.usp.pqi.process.analysis`

Implementing PROCESS CONTROL INTERFACE

Another artifacts breed was conceived to implement functionalities grouped under the PROCESS CONTROL INTERFACE component. It was identified as *Control Panel*. Unlike *Tags Board* breed, there will be just one instance of *Control Panel* for each section of the plant where RTO will be applied.

Control Panel has the mission to keep in touch with the plant control structures and apply the results of RTO cycles as set-point updates. Another functionality of this artifact is to perform checks on the feasibility of proposed set-point updates. To do that, it creates a chain of internal checkers, which are instances of **IFeasibilityChecker** interface from RTOF, and run them in a row. The set-points' values are declared feasible if all the checkers say the same about it. In that process *Control Panel* can access properties and methods of *Tags Board* instances.

Observable properties implemented in *Control Panel* are:

connected: Says if the artifact has communication with the plant control system.

Operations include:

setupControlPanel: Used to setup the artifact, passing all necessary parameters. A list of feasibility checkers is one of them.

checkOptimization: Checks a set of set-point's values and returns **True** in case all checkers return same value.

updateSetPoints: Used to ask *Control Panel* to update plant set-points. It will try to apply changes and call the necessary routines of the control system.

Along with CArtAgO classes, *Control Panel* artifacts use the following RTOF packages:

- `br.usp.pqi`
- `br.usp.pqi.process`
- `br.usp.pqi.process.optimization`

7.1.2 Implementing OPTIMIZER component

As the architectural view represented in figure 4.4 shows, the reactive character of OPTIMIZER gets implemented in the integration of two subcomponents. The first, called IMPROVER, acts as an intentions engine —i.e, it reads the environment and other inputs and produces intentions to improve process's performance. The subcomponent called ACTUATOR got the responsibility of choosing which one of the candidate intentions will be passed to the next layer by means of **Control Panel** artifact. For that sake it runs a decision mechanism. Interaction between both components occurs using a connector of type Shared Repository.

According to the responsibilities and functions of each of the three elements of OPTIMIZER, it stood clear that, due to their reactive nature, IMPROVER and ACTUATOR should be implemented using agents as central entities and some artifacts to help them to do the work. By its part, the connector should be an artifact. Next sections shed more light about the implementation.

Implementing IMPROVER

As described in functional requirement **FR-5** and in section 4.2.3, the RTO system must allow the run of several RTO approaches concurrently. That is why IMPROVER was conceived to be implemented as a set of agents, each of them running a different approach, or the same approach with some variation, in order to propose candidate solutions for the RTO problem.

Two agents were implemented in the prototype instancing components IMPROVER A and IMPROVER B of the architectural section showed in figure 4.6. Each of them running the RTO approach known as MPA, with some variation in their parameters.

As the EMSOC command line tool had been already developed to serve as simulation and optimization engine, it was wrapped as a CArtaGO artifact. In that way, the *EMSOC* artifact could be used by IMPROVER agents as a device to simulate models, estimate parameters and solve optimizations.

Observable properties implemented in *EMSOC* are:

state: Says if EMSOC artifact is idle or solving some problem.

flowsheet: Shows the flowsheet type of the problem been solved (Simulation, Parameters Estimation or Optimization).

Operations:

simulate: Runs a simulation, expecting as parameters the EMSO flowsheet path and the simulation name.

solve_estimation: Runs a parameters estimation, expecting the EMSO flowsheet path and the problem name as input parameters.

solve_optimization: Solves an optimization problem, expecting as parameters the EMSO flowsheet path and the problem name.

Following the input philosophy of EMSO, a Flowsheet, a Parameters Estimation and an Optimization file were generated by the project team, using the EML language. These files are detailed in appendixes E, F and H. Experimental data resulted from observations and used in parameters estimation was collected in a file with specific format (appendix G).

In order to tackle the described constraints and to take advantage of the model, flowsheet and problem files developed for REPLAN, the metaphor introduced in CArtaGo was used again to implement two more artifacts: one for wrapping the concept of *EMSO Optimization Flowsheet* and other for the *EMSO Data File*. As an EMSO model can be embedded or referred from a flowsheet file, there was no need to represent a model as an artifact.

EMSO Optimization Flowsheet's properties are:

file_path: Shows the optimization problem file path.

It has a single operation:

updateModelParameters: Called when a solution from the parameters estimation problem was found and the new values need to be updated for the new MPA iteration.

EMSO Data File artifact exhibits following properties:

file_path: Shows the path to the experiments file.

experiments_count: Shows how many experiments the file contains.

max_experiments_count: Shows the actual limit for experiments count in the file. Experiments can be added while the RTO iterations runs and new steady-states periods are detected.

It offers a single operation:

saveObservedData: Used when new data needs to be saved to the file.

Implementing SHARED REPOSITORY connector

For the interaction between IMPROVERS and ACTUATOR, as defined by the architecture, a Shared Repository connector was selected. Following that idea an *Optimizations Stage* artifact was implemented. Whenever an IMPROVER instance finds a possible optimization for the process, it will deploy it at the *Optimizations Stage* and continue to its next iteration. From that point, it is ACTUATOR's responsibility to decide what to do

with the staged element.

Properties of *Optimizations Stage*:

candidates_count: Says at anytime how many candidates solutions are in the stage.

Operations:

stageOptimization: Called by an agent to stage a new candidate optimization. The agent identity and the expected profit if the solution is applied, are also passed in the operation call.

pollOptimizations: Polls the elements in the stage removing them.

Implementing ACTUATOR

Functional requirement **RF5** motivated also having the architectural component ACTUATOR, this way centralizing decisions on which of the suggested process improvements should be passed to the *Control Panel* artifact as set-points update requests. That induced to implement ACTUATOR as a single agent that uses an specific artifact, called *Improvers Ranking*, to help it in the decision taking process. As described in section 4.2.3 that procedure is referred as POP.

Improvers Ranking artifact serves as a statistics repository for every IMPROVER agent that, at some point, deployed a candidate improvement in *Optimizations Stage*. For each registered agent, *Improvers Ranking* keeps a set of performance counters, implemented using the **IPerformanceCounter** interface from RTOF. Each counter is identified by a name and has a float value associated with it. The idea is to use these counters as performance measures of IMPROVER instances. Along with information from PROCESS INFORMATION SERVICE component and its *Tags Board* artifacts, ACTUATOR agent can check agents' counters, whenever it considers necessary, to perform POP at each time. ACTUATOR agent has also the responsibility to keep *Improvers Ranking* updated.

Properties implemented in *Improvers Ranking*:

counters_count: Says at anytime how many counters are kept for each IMPROVER agent.

Operations implemented in *Improvers Ranking*::

setupRanking: Used to setup the artifact, passing all the necessary parameters. A list of counters is expected.

registerAgentInRanking: Registers an agent by its identity in the artifact.

incrementCounter: Increments an specific counter for some agent. The agent's identity and the counter name is expected. The counter is incremented by one by default but this operation has a variant where the value to be incremented can also be specified.

getCounterValue: Returns the actual value of an specific counter for an agent. The agent identity and the counter name are expected to be passed.

Functional requirement FR-6 motivated the creation of *Optimization Log* artifact, that serves as a binnacle to register all the improvements proposed by IMPROVER agents during system life and which of them were actually sent to the plant control system. Once a candidate is sent to the *Control Panel* artifact, ACTUATOR gets a feedback about if the set-points update was actually sent or was rejected for some reason. ACTUATOR then updates the *Optimization Log* with that information.

Properties implemented in *Optimization Log*:

optimizations_count: Says at anytime how many optimizations have been received so far.

rejected_optimizations_count: Says at anytime how many optimizations have been received and rejected so far.

Operations implemented in *Improvers Ranking*:

log_optimization: Used to log a new element, passing all the details.

update_optimization: Used to update some data about an element that is already logged.

As explained in section 4.2.3, POP is triggered according to the poll strategy that is active inside ACTUATOR. As already exposed, POP have two phases: POP-DNF, where the discard of non feasible improvements occurs, and POP-CB, where a decision is made about which candidate improvement will be chose if more than one was staged. In practice, POP happens thanks to the interaction of ACTUATOR with *Tags Board*, *Control Panel*, and *Improvers Ranking* artifacts, as it will be explained in the next sections.

7.1.3 RTOMAS overall view and dynamic

This section pursues to give an idea of the internal dynamic of **RTOMAS** as a whole. Figure 7.2 shows an overall view of a possible running scenario of **RTOMAS**, containing

Figure 7.2 – RTOMAS overall view

agents and artifacts interacting. Arrows connecting elements represent interactions between them and specifies which architectural data element is transferred.

A principle followed in the implementation was that every agent, during initialization, should get ready the artifacts it will need to interact with. In case of artifacts shared by several agents, one of them should take the responsibility to set it up.

From a bird's eye view, ACTUATOR architectural component can be considered the central arm of the system. After all, it is from ACTUATOR where plant improvement requests are triggered. On the other hand, the system's "brain" is spread between agents and artifacts. IMPROVER agents source candidate solutions feeding ACTUATOR through *Optimizations Stage* with options to act. *Tags Board*, *Control Panel*, and *Improvers Ranking* assist ACTUATOR to "make its mind" about the next action.

Following the prior idea, first element instantiated at run time is ACTUATOR agent. During its initialization ACTUATOR creates and setups *Tags Board*, *Optimizations Stage*, *Control Panel*, *Improvers Ranking* and *Optimization Log* artifacts. ACTUATOR must provide to them all the necessary information to initialize. Once the agent has created artifacts it will keep observing *Tags Board* events and listening for staging actions in the *Optimizations Stage*. When it receives an alert that a new element was staged, the POP procedure is triggered, depending on the poll strategy that is active. When a decision is made, the candidate solution is sent to *Control Panel* as set-points update request.

In order to send the final order to the plant control system, *Control Panel* can check the process status by means of *Tags Board* instances. If it finds some reason to reject the candidate, ACTUATOR is notified about that and, as already mentioned, the *Optimizations Log* artifact is updated.

IMPROVER agents are created after ACTUATOR, instancing as many as different RTO approaches or variants will be running. As already mentioned, two instances were defined for the REPLAN prototype, both running MPA approaches. Small differences in the nonlinear programming solvers' parameters, declared in the EML optimization files, were introduced to induce differences in the solutions coming from both agents.

IMPROVERS should create necessary artifacts that have not been instantiated yet. In this case, they depend on EMSO to figure out solutions, so they will setup instances of *EMSOC*, *EMSODataFile* and *EMSOOptFlowsheet* artifact breeds. Once IMPROVER instances are ready they start to observe *Tag Board* events, ask for data, and stage possible improvements into the *Optimizations Stage* artifact.

Next listing shows the JaCaMo main source file, declaring agent instances and order of creation:

```
mas MAS4RTO {

    agent ag.ACTUATOR    : actuator.asl

    agent ag.MPA_OPTIMIZER    : mpa_optimizer.asl
    agent ag.MPA_OPTIMIZER_1 : mpa_optimizer_1.asl

    // agent source path
    asl-path: src/agt
    src/agt/inc

    class-path: "/usr/local/lib/jdbc/postgresql-9.2-1003.jdbc4.jar"
               "/usr/local/lib/joda-time/2.9.4/joda-time-2.9.4.jar"
}
```

In a real application of the MAS, new IMPROVER agents can join the system at run time, bringing new approaches, perspectives and variability into it. The only constraint is that they must follow the rules and roles defined by the MAS Organization.

7.1.4 Implementing agents' mind

One of the most interesting challenges of this work, was to figure out how to implement RTO and some of its approaches —e.g, MPA— using concepts associated to BDI agents, following the proposed system architecture and using the Jason language as the final construction blocks. The purpose of this section is to describe briefly how the those elements and the resources they provide were used.

ACTUATOR' mindset

As already mentioned, ACTUATOR agent can be considered the central active element of the system. With that in mind, some basic concepts and data were modeled as ACTUATOR's initial believes, using *predicates* and *structures*, from the Jason's repertory. That is the case of:

- **Process Tags:** Metadata of process variables that were going to participate in the RTO run, in the form of key-value pairs. For each Tag it was declared an *ID*, a *Variable Name*, a *Description* and a *Label*.
- **Tags Data Source:** Metadata about how *Tags Board* artifact connects to the plant control system, in order to read Tag's values at real time. Elements include the *URL* of the data source, and a *User* and *Password* to grant access. For the prototype purposes, a PostgreSQL table was used as a simulation of data source. That is why there is a *Dataset* element in the metadata, referencing the table name inside the database used as fake data source.
- **Steady-State Detection Problem:** The steady-state detection was modeled as a problem and its most important details were declared as metadata. It includes the Tags that will be used to identify stability, the Sampling and Detection Frequency, the Time Windows Size, the Detection Method, among others.
- **Set-points:** Metadata about the process Tags that will act as set-points.
- **Poll Strategy:** Which is the active strategy for triggering the POC procedure. The options are:
 - **first_available:** Starts POC as soon as a candidate is deployed at *Optimizations Stage*.
 - **timer_since_first_available:** Once a candidate is staged, ACTUATOR starts a timer to wait for other candidates. If the timer reaches its limit actual elements in the stage will be used in the POC and polled from the stage.

- **frequent_checking**: ACTUATOR performs checks at regular intervals to the *Optimizations Stage*. On each check actual candidates are polled and used.
- **Feasibility Checkers**: Metadata about the checkers to be used in the *Control Panel* artifact to check the feasibility of candidate optimizations.
- **Performance Counters**: Metadata about the counters to be used in the *Improvers Ranking* artifact, to measure the IMPROVER agents performance.

Next listing shows ACTUATOR's initial beliefs and rules containing specific metadata for the REPLAN distillation unit.

```

/***** Initial beliefs and rules *****/

process_tags (
    [
        tag( id("fs_f_feed_p"),      var("FS.F_feed_p"),      desc("Feed
            flow rate [kg/h]"),      label("FS.F_feed_p")    ),
        tag( id("fs_pb_mass"),      var("FS.PB_mass"),      desc("Botton
            flow rate [kg/h]"),      label("FS.PB_mass")    ),
        tag( id("fs_d_mass"),      var("FS.D_mass"),      desc("
            Distillate flow mass base"),      label("FS.D_mass")    ),
        tag( id("fs_f_new_z_2"),      var("FS.F_new.z(2)"),      desc("
            Concent de propileno na entrada"),      label("FS.F_new.z(2)") ),
        tag( id("fs_pb_z_2"),      var("FS.PB.z(2)"),      desc("
            Concent de propileno no fundo"),      label("FS.PB.z(2)")    ),
        tag( id("fs_lreflux_mass"), var("FS.Lreflux_mass"), desc(""),
            label("FS.Lreflux_mass") ),
        tag( id("fs_fcout_mass"),      var("FS.FCOut_mass"),      desc(""),
            label("FS.FCOut_mass")    ),
        tag( id("fs_fbin_mass"),      var("FS.FBIn_mass"),      desc(""),
            label("FS.FBIn_mass")    ),
        tag( id("fs_f_mass"),      var("FS.F_mass"),      desc(""),
            label("FS.F_mass")    ),
        tag( id("fs_propa_top"),      var("FS.Propa_top"),      desc(""),
            label("FS.Propa_top")    ),
        tag( id("fs_f_new_z_3"),      var("FS.F_new.z(3)"),      desc(""),
            label("FS.F_new.z(3)")    ),
        tag( id("fs_ltray_17_t"),      var("FS.Ltray(17).T"),      desc(""),
            label("FS.Ltray(17).T")    ),
        tag( id("fs_ltray_35_t"),      var("FS.Ltray(35).T"),      desc(""),
            label("FS.Ltray(35).T")    ),
        tag( id("fs_ltray_51_t"),      var("FS.Ltray(51).T"),      desc(""),
            label("FS.Ltray(51).T")    ),
        tag( id("fs_ltray_69_t"),      var("FS.Ltray(69).T"),      desc(""),
            label("FS.Ltray(69).T")    ),
    ]
)

```

```

tag( id("fs_ltray_85_t"),    var("FS.Ltray(85).T"),  desc(""),
     label("FS.Ltray(85).T") ),
tag( id("fs_ltray_119_t"),  var("FS.Ltray(119).T"), desc(""),
     label("FS.Ltray(119).T") ),
tag( id("fs_ltray_137_t"),  var("FS.Ltray(137).T"), desc(""),
     label("FS.Ltray(137).T") ),
tag( id("fs_ltray_153_t"),  var("FS.Ltray(153).T"), desc(""),
     label("FS.Ltray(153).T") ),
tag( id("fs_ltray_171_t"),  var("FS.Ltray(171).T"), desc(""),
     label("FS.Ltray(171).T") ),
tag( id("fs_p_top"),        var("FS.P_top"),        desc(""),
     label("FS.P_top")      ),
tag( id("fs_p_bot"),        var("FS.P_bot"),        desc(""),
     label("FS.P_bot")      ),
tag( id("fs_twater_out"),   var("FS.Twater_out"),   desc(""),
     label("FS.Twater_out") ),
tag( id("fs_vboiler_t"),    var("FS.Vboiler.T"),    desc(""),
     label("FS.Vboiler.T")  )
]
).

tags_datasource (
    url("jdbc:postgresql://localhost/rto"),
    user(rtouser),
    password(rtoadmin),
    dataset(replan_tags)
).

ss_detection_problem (
    tags(["fs_f_feed_p", "fs_pb_mass", "fs_d_mass", "fs_f_new_z_2",
          "fs_pb_z_2"]),
    time_unit(seconds),
    sampling_start_delay(2),
    sampling_freq(5),
    time_windows_size(5),
    ss_detection_start_delay(10),
    ss_detection_freq(10),
    ss_detection_method(random),
    ss_detection_use_recent_date(true)
).

instant_profit_calculation (
    class("br.usp.pqi.process.sampling.SumAllProfitFormula")
).

setpoints(
    [

```

```

        tag( id("fs_lreflux_mass"), var("Lreflux_mass")),
        tag( id("fs_pb_mass"), var("PB_mass_p"))
    ]
).

/-- OOP Poll Strategies -//
poll_strategy(first_available).

/-- Feasibility Checkers -//
feasibility_checkers(
    [
        checker(name("OOP_SPs_Distance_Checker"), class("br.usp.pqi.
            process.optimization.SetPointDistanceChecker"), desc(""))
    ]
).

performance_counters(
    [
        counter(name("staged_opts"), desc("Staged Optimizations count"))
        ,
        counter(name("dnf_opts"), desc("Optimizations discarded in DNF
            phase"))
    ]
).

```

As showed in *tags_datasource* belief definition, a PostgreSQL database was used as simulation of the plant information system, providing data for all the *Tags* participating in the RTO run.

Next listing shows another section from ACTUATOR's source code detailing the agent's goals. As it stays clear, ACTUATOR has a single goal in mind: to perform RTO. That was modeled directly using a Jason goal declaration: `!do_RTO`. To target that goal, several plans were defined, all lead by a central plan, which launches new subgoals.

```

/***** Initial goals *****/

!do_RTO.

/***** Plans *****/

+!do_RTO : true

    <-      .my_name(MyName);
           .print(MyName ," starting ...");

```

```

!environment_ready;
!observe_tags_board;
!listen_for_optimizations
.

```

As it can be seen, first subgoal of ACTUATOR is to get the environment ready, which is translated into setting up all the necessary artifacts. After that it wishes to observe the tags from the *Tags Board* artifact. Finally it aims to listen for candidates optimizations. Implementation details of plans for subgoals are not given in this section. The complete source code for ACTUATOR can be found in the Appendixes section of this work.

A source code section that is useful to better understand how ACTUATOR's mindset looks like is listed below. It shows the implementation of a plan for when a new candidate optimization was staged by some IMPROVER and the active poll strategy is `first_available`.

```

+optimization_staged(Optimization) : poll_strategy(first_available)

    <-      cartago.invoke_obj(Optimization, getSource,
        SourceAgentID);
        .print("Optimization staged by agent ", SourceAgentID);
        incrementCounter("staged_opts", SourceAgentID);

        checkOptimization(Optimization, Is_Feasible);
        .print("Optimization checked");

        if (Is_Feasible) {

            .print("A feasible Optimization was found by
                agent ", SourceAgentID);
            cartago.invoke_obj(Optimization, getVarValues,
                Var_Values);
            !update_setpoints(Var_Values);
        }
        else {

            .print("The Optimization found by agent ",
                SourceAgentID, " is not feasible to apply.");
            incrementCounter("dnf_opts", SourceAgentID);
        }

.

```

Once a new candidate was deployed at the *Optimizations Stage* ACTUATOR update the counter `staged_opts` for the source agent. As described in a prior section, the `incrementCounter` action is an available operation of *Improvers Ranking* artifact. After that the agent check if the candidate is a feasible solution by means of the *Control Panel*'s operation `checkOptimization`. If the candidate was found feasible ACTUATOR acts again through *Control Panel* to ask for a set-points update. If the optimization was not feasible the IMPROVER's performance is updated incrementing `dnf_opts` counter. Just before asking the *Control Panel* for updating set-points, a check about the plant state is done. The next listing shows that check in the form of agent's plans with conditions to be triggered.

```
+!update_setpoints(Var_Values) : process_state("steady")

  <- .print("I do can update set-points as process is steady");
    updateSetPoints(Var_Values);
.

+!update_setpoints(Vars) : process_state("transient")

  <- .print("Was going to update set-points by process is transient");
.
.
```

IMPROVER' mindset

Concepts modeled as IMPROVER's initial believes, using Jason's predicates and structures were:

- Estimation problem: Metadata about the parameters estimation problem.
- Optimization problem: Metadata about the optimization problem.
- EMSOC tool: Path to the EMSOC tool executable file.
- RTO step: Initial phase of the RTO cycle.

The listing below shows the actual Jason code with the mentioned elements. As it can be verified, there are metadata elements pointing to actual EMSO input files. *Est_REPLAN.mso* and *Opt_REPLAN.mso* (appendixes F and H) contain the estimation and optimization problems definition in EML respectively. *Flow_REPLAN.mso* (appendix E) contains the flowsheet representing the distillation unit structure and *Data_REPLAN.dat* (appendix G) collect observations from the plant when in steady-state.

```

/***** Initial beliefs and rules *****/

estimation_problem (

    file("/home/elyser/Temp/RT0/eml/Est_REPLAN.mso"),
    name("Col_Est"),
    data_file("/home/elyser/Temp/RT0/eml/Data_REPLAN.dat"),
    params(["Eir2", "Eir1", "Eis", "Propy_p", "F_feed_p", "
        Lref_mass_p", "PB_mass_p", "P_top",
        "P_bot", "QTow", "vfB", "U_cooler"])
).

optimization_problem (

    file("/home/elyser/Temp/RT0/eml/Opt_REPLAN.mso"),
    flowsheet("/home/elyser/Temp/RT0/eml/Flow_REPLAN.mso"),
    name("Col_Opt"),
    free_vars(["Lreflux_mass", "PB_mass_p"])
).

emsoc_tool (

    path("/home/elyser/Temp/RT0/eml/emsoc")
).

rto_step(ss_detection).

```

In spite of the fact that each IMPROVER agent can implement a different approach, some common constructions were identified while implementing the prototype. All IMPROVER instances will have a unique and common initial goal: to get the process optimized. To target that goal, a general plan can be defined, containing essential steps: observe process tags, get the environment ready and to register in the ranking.

```

/***** Initial goals *****/

!process_optimized.

/***** Plans *****/

+!process_optimized : true

    <- .my_name(MyName);
    .print(MyName , " starting ...");

```

```

!rto_step_printed;
!observe_tags_board;
!environment_ready;
!register_in_ranking;
.

```

MPA agents can be considered as a breed which instances will act according to MPA steps. That's why the source code for that breed contains a plan for an event that is triggered when a signal is received from Tags Board claiming that an steady-state was detected. Next listing shows that code.

```

+in_steady_state(Rep_State) : rto_step(ss_detection)

  <- .print("steady-state detected !!!");

  saveObservedData(Rep_State, false);

  -+rto_step(params_estimation);
  !rto_step_printed;

  ?estimation_problem(
    file(File),
    name(Problem),
    data_file(_),
    params(Params));

  stopSampling;
  solve_estimation(File, Problem, Params);
.

```

When the event that indicates a new steady-state detected signal was emitted occurs, plan showed in the listing is triggered. First notable action of that plan is to save the calculated representative steady-state as a new experiment using an operation from *EM-SODataFile* artifact. Next the agent updates its mental state getting to the next MPA phase: parameters estimation. After that the agent appeals to its “memory” to retrieve details about the parameters estimation problem, and pass to solve it, by means of an operation of *EMSOC* artifact. In the process ACTUATOR sends an order to *Tags Board* artifact to stop sampling.

Next listing corresponds to the plan that is triggered when the agent is notified that estimation problem was solved.


```

+estimation_solved(Estimated_Params) : true

  <- .print("Model's parameters were estimated: ", Estimated_Params);

  updateModelParameters(Estimated_Params);

  -+rto_step(optimization);
  !rto_step_printed;

  ?optimization_problem (
    file(File),
    flowsheet(_),
    name(Problem),
    free_vars(Free_Vars))

  solve_optimization(File, Problem, Free_Vars);

.

```

First notable action updates plant model's parameters with new values found during estimation. That occurs thanks to an operation available in *EMSOOptFlowsheet* artifact. Next step is to update agent's mental state switching to the optimization phase. Again the agent appeals to its mind to bring into the context information about the optimization problem and goes after solving it, using an operation from the *EMSOC* artifact.

Once a notification indicating a solution was found for the optimization problem MPA IMPROVER runs the plan showed next:

```

+optimization_solved(Free_Var_Values, Promised_Profit) : true

  <- .print("New Optimization was calculated: ", Free_Var_Values);
  .my_name(MyName);

  stageOptimization(MyName, Promised_Profit, Free_Var_Values);

  -+rto_step(ss_detection);
  !rto_step_printed;

  startSampling;

.

```

As soon as the agent has the optimization problem solution in hands, it stages a new object the *Optimizations Stage* artifact. That object, a new candidate to be turned out

into set-points update, contains values for the free variables and also a promise of profit if the values are applied to the plant control system. At the end IMPROVER just returns to initial RTO phase: steady-state detection and restarts to sample the plant.

7.1.5 RTOMAS as a predefined organization of agents

As a predefined agents organization, the designed system defines some roles and rules for the on-line optimization implementation. The organizational rules are:

- There can be several instances of *Tags Board* artifact and their properties and operations can be accessed by any agent breed.
- All agents and artifacts that need to get information and data from the process must do it through *Tags Board* artifacts.
- There can be several instances of the IMPROVER breed, been very well defined their role inside the organization: to propose candidate RTO solutions.
- IMPROVER instances have their action space limited to deploy the candidate solutions to the *Optimizations Stage* artifact.
- There must be only one instance of *Optimizations Stage* artifact.
- There must be only one instance of *Improvers Ranking* artifact.
- There must be only one instance of ACTUATOR breed.
- Just ACTUATOR can read from *Optimizations Stage* and *Improvers Ranking* artifacts.
- All the communication with the plant control system must be done through *Control Panel* artifact.
- There must be only one instance of *Control Panel* artifact.
- Just the ACTUATOR can communicate with the *Control Panel* artifact.
- In the case of distributed optimization there could exist more than one organization instance, applying all the former rules to each instance.

7.1.6 Running RTOMAS

The developed prototype was run in a single machine and in a test environment —i.e., not in a real plant— because of security and operational limitations.

The JaCaMo platform provides by default a GUI as a console for the MAS. Each agent gets a tab in the console, where its outputs are showed. There is a tab identified as *common* that shows outputs from all agents. The GUI can be used also to pause, stop or debug the application. Figures 7.3 to 7.6 shows the MAS running and some output from its components.

Figure 7.3 shows the GUI with the *common* tab selected. The log shows the system starting and the agents doing their first actions. Agents MPA_OPTIMIZER, MPA_OPTIMIZER_1 and ACTUATOR are the first entities to be built, creating and focusing on the artifacts that are necessary for their role —i.e., the *Tags Board*, *Optimizations Stage*, *Control Panel*, *Improvers Ranking*, *EMSO Optimization Flowsheet* and *EMSO Data File*. MPA_OPTIMIZER and MPA_OPTIMIZER_1 get registered in the *Improvers Ranking* and start in steady-state detection phase. The *Tags Board* artifact start to sample and analyze the plant.

Figure 7.3 – RTOMAS console. General system output.

Figure 7.4 shows the log coming just from MPA_OPTIMIZER. At some point in the log, the agent jumps to the parameter estimation phase. When finished the estimation, the resulting parameters are detailed. Next line of the log indicates agent entered into the optimization phase. Once the problem is solved, the free variable values are logged and another RTO iteration starts over.

Output from agents MPA_OPTIMIZER and MPA_OPTIMIZER_1, at the time when both are in the parameters estimation phase, and they have called their respective *EMSOC* artifacts to solve the problem, is showed in figure 7.5. Actually, the log is nothing else than

Figure 7.4 – RTOMAS console. OPTIMIZER agent output.

the direct output from the *EMSOC* instances. The Rotational Discrimination algorithm [180] was used for parameters estimation. The RCOMPLEX algorithm [181] was used for solving the optimization.

Figure 7.5 – RTOMAS console. EMSOC artifacts in action.

Events triggered inside the ACTUATOR agent are showed in its specific tab of the GUI, as figure 7.6 details. Candidates improvements are staged in the *Optimizations Stage* by MPA_OPTIMIZER and MPA_OPTIMIZER_1. Following the *first_available* poll strategy, once the ACTUATOR gets aware of each of them, the Pick Out Procedure (POP) is run.

Whenever the feasibility of the candidate is verified and since the plant is still stable, a request to update set-points is sent to *Control Panel*. At some point in the log, ACTUATOR discarded a candidate improvement coming from MPA_OPTIMIZER_1 due to its non-feasibility.

Figure 7.6 – RTOMAS console. ACTUATOR output

An snapshot taken during the optimization phase of one of the RTO iterations is showed in figure 7.7. The solver converged after 101 iterations. The whole RTO cycle, from the detection of the steady-state to sending the set-points update request, took 2 minutes, 21 seconds and 57 milliseconds. The machine used for testing runs a 64-bit Ubuntu 12.04 operational system, and features an Intel® Core™ i7-3632QM processor, at 2.20 GHz, with 6 GiB of main memory.

At run time, JaCaMo also provides a service that allows to make requests to an http server in order to get information about agents internal state at each moment. Similar information can be checked for artifacts of the environment. Figures 7.8 to 7.11 show the use of that resource.

```

MAS Console - mAS4RTO
common [EMSOC] If condition:FS.Propa_top.>5000 switched to:1
ACTUATOR [EMSOC] If condition:FS.PB.z(2)<0.003 switched to:0
MPA_OPTIMIZER [EMSOC] If condition:FS.Propa_top.>5000 switched to:0
[EMSOC] Objectives: -36474.8
[EMSOC] Iteration: 101
[EMSOC] x[0] = 3.13639
[EMSOC] x[1] = 6.52042
[EMSOC] S(x) = -36474.8
[EMSOC] -----
[EMSOC] Iterations count: 101
[EMSOC] x[0] = 3.13639
[EMSOC] x[1] = 6.52042
[EMSOC] S(x) = -36474.8
[EMSOC] -----
[EMSOC] NLP solver converged.
[EMSOC] Objective function: 191
[EMSOC] Equality constraints: 0
[EMSOC] Inequality constraints: 0
[EMSOC] Objective function Jacobian: 0
[EMSOC] Equality constraints Jacobian: 0
[EMSOC] Inequality constraints Jacobian: 0
[EMSOC] Optimization of 'Col_Opt' finished successfully
[EMSOC] DestroyJavaVM...done
[MPA_OPTIMIZER] New Optimization was calculated: [map("Lreflux_mass",364.34),map("PB_mass_p",-1)]
[ACTUATOR] Optimization staged by agent ag.MPA_OPTIMIZER
[OOP_SPs_Distance_Checker] Checking SetPoint distances...
[MPA_OPTIMIZER] RTO Step: ss detection
[OOP_SPs_Distance_Checker] Distance for tag fs_lreflux_mass = 5.660003662109375. Threshold = 20.0
[OOP_SPs_Distance_Checker] Distance for tag fs_pb_maas = 16.03437614440918. Threshold = 20.0
[CONTROL_PANEL] Checker OOP_SPs_Distance_Checker says YES
[ACTUATOR] Optimization checked
[ACTUATOR] A feasible Optimization was found by agent ag.MPA_OPTIMIZER
[ACTUATOR] I do can update set-points as process is steady
[CONTROL_PANEL] Updating setpoints
Lreflux_mass = 364.34
PB_mass_p = -1.0
Clean Stop Pause Debug New agent Kill agent New REPL agent Sources

```

Figure 7.7 – RTOMAS console. Optimization converged after 101 iterations.

```

Jason Mind Inspector - ...
127.0.1.1:3272
Agents
- ag ACTUATOR
- ag MPA_OPTIMIZER
- ag MPA_OPTIMIZER_1
by Jason
Inspection of agent ag.MPA_OPTIMIZER
- Beliefs
connected(true){antifact_id(cobi_2),percept_type(obs_prop),source(percept),antifact_name(cobi_2.TAGS_BOARD),workspace(cobi_2.main.cobi_0)}
emsoc_tool(path("/home/elyser/Temp/RTO/eml/emsoc")){source(self)}
estimation_problem(file("/home/elyser/Temp/RTO/eml/Est_REPLAN.mso"),name("Col_Est"),data_file("/home/elyser/Temp/RTO/eml/Data_REPLAN.dat"),params(["Eir2","Eir1","Eis","Propy_p","F_feed_p","Lref_mass_p","PB_mass_p","P_top","P_bot","Q_Tow","vB","U_cooler"])){source(self)}
flowsheet("NONE"){antifact_id(cobi_6),percept_type(obs_prop),source(percept),antifact_name(cobi_6.EMSOC),workspace(cobi_6.main.cobi_0)}
focused(main.EMSOC){antifact_type("br.usp.pqi.ro.art.EMSOC"),cobi_6}{source(percept)}
focused(main.TAGS_BOARD){antifact_type("br.usp.pqi.ro.art.TagBoard"),cobi_2}{source(percept)}
instant_profit(0){antifact_id(cobi_2),percept_type(obs_prop),source(percept),antifact_name(cobi_2.TAGS_BOARD),workspace(cobi_2.main.cobi_0)}
joined(main.cobi_0){antifact_id(cobi_1),percept_type(obs_prop),source(percept),antifact_name(cobi_1.ag.MPA_OPTIMIZER-body),workspace(cobi_1.main.cobi_0)}
last_analysis_time(cobi_27){antifact_id(cobi_2),percept_type(obs_prop),source(percept),antifact_name(cobi_2.TAGS_BOARD),workspace(cobi_2.main.cobi_0)}
last_sampling_time(cobi_8){antifact_id(cobi_2),percept_type(obs_prop),source(percept),antifact_name(cobi_2.TAGS_BOARD),workspace(cobi_2.main.cobi_0)}
optimization_problem(file("/home/elyser/Temp/RTO/eml/Opt_REPLAN.mso"),flowsheet("/home/elyser/Temp/RTO/eml/Flow_REPLAN.mso"),name("Col_Opt"),free_vars(["Lreflux_mass","PB_mass_p"])){source(self)}
process_state("transient"){antifact_id(cobi_2),percept_type(obs_prop),source(percept),antifact_name(cobi_2.TAGS_BOARD),workspace(cobi_2.main.cobi_0)}
rto_step(optimization){source(self)}
sampling(true){antifact_id(cobi_2),percept_type(obs_prop),source(percept),antifact_name(cobi_2.TAGS_BOARD),workspace(cobi_2.main.cobi_0)}
state("IDLE"){antifact_id(cobi_6),percept_type(obs_prop),source(percept),antifact_name(cobi_6.EMSOC),workspace(cobi_6.main.cobi_0)}
Annotations

```

Figure 7.8 – RTOMAS web interface. Inspection of OPTIMIZER agent.

Figure 7.9 – RTOMAS web interface. Inspection of ACTUATOR agent.

Figure 7.10 – RTOMAS web interface. Inspection of TAGS BOARD artifact.

Figure 7.11 – RTOMAS web interface. Inspection of OPTIMIZERS RANKING artifact.

8 DISCUSSION

This chapter discusses ideas related to what we consider as benefits derived from the presented work. To organize the discussion, it was segmented according to the main contributions.

8.1 A software architecture

A central contribution of this work was to carry out a survey of functional and non-functional requirements for a generic RTO system, and based on them, to design a software architecture that can fulfill those requirements.

Counting on a software architecture description for RTO systems brings several benefits:

- Starting from an architecture is worth since an RTO system can be implemented using several technologies and paradigms compatibles with that architecture, but its main functionalities and quality attributes will be kept.
- The system and RTO method can be easier to understand, as functional requirements are assigned to specific units of functionality. Interactions between components and properties associated to them are also part of the architecture description. The fact that the system has an internal organization, and interactions between system's elements are clear, incentives brainstorming.

This will help the academy and the industry to share a common vision of how an on-line optimization system performs its work, and where the several RTO concerns can be tackled, both at design and run time.

- As, after all, an architecture restricts the vocabulary of design alternatives for a system, so it channels the creativity of RTO developers and implementers, reducing that way the system complexity.
- Decisions made in an architecture allow developers and researchers to reason about changes needed as the RTO system evolves, and how to manage them. That can be decisive for the success of the system and RTO itself, especially when dealing with software that is in operation at a real production unit.
- The architecture provides the basis for allowing incorporation of independently developed RTO software components.

- A documented architecture enables communication between stakeholders and system users. That, traduced to a real plant, means a better communication between operators, engineers and managers.
- A documented architecture is a key artifact to reason about the cost of an RTO implementation. It could help as well in the schedule of new research and development actions.
- An architecture for RTO systems can be the seed for a new research field that could have as target the creation of software components totally oriented to on-line optimization. That way, future RTO applications could be developed just assembling those components, rather than creating from scratch.
- An architecture can be used for training new system operators, as well as process engineers and managers.

The fact of implementing the RTO prototype following the designed architecture brought the following benefits:

- A well organized and structured software, improving the understanding and maintainability of its functional parts.
- The possibility to distribute the RTO functional steps through the hardware infrastructure, which brings increment of parallelism and independence between them. Although performance of an RTO application can be limited by the solver used in estimation and optimization phases, as well as by the complexity and size of the optimization problem, the constraints applied during the architecture design helped to avoid additional performance issues.
- The introduction of generic software components, which can be customized to approach RTO tasks in specific ways, keeping the application skeleton simple and immutable. All this bring benefits in terms of re-usability of the application in several context and environments.

8.2 A software framework for RTO

Having the possibility to start from concepts and ideas already implemented as elements of a software framework, speeds up the implementation of new RTO systems. Using the provided software features or extending them, users can adapt the framework to specific situations.

Having abstract entities modeling several concepts can be of great help to the evolution and better understanding of RTO as a field and its ontology. At the same time, developers and researchers can focus, since early work stages, on more specific concerns, contributing to the performance of research actions.

8.3 A workbench for RTO research

A direct contribution of **RTOMAS** is that, due to the way it was designed and implemented, it serves as a rich workbench where new concepts, ideas and hypothesis about RTO can be developed and tested.

The system's structure makes it easy to test several paths and to challenge well-accepted techniques. An example is that a reactive or proactive checking strategy to test the steady-state condition of the plant can be implemented. The developed prototype used a reactive check as part of MPA approach, as it waits for an event indicating plant is stable; but it could be implemented as a proactive action, where agents will check plant state at regular intervals.

8.3.1 Approaching global and distributed optimization

According to global and distributed optimization approaches, the large-scale plant can be visualized as a unique process or be decomposed into subsystems in order to reduce complexity.

For global optimization, a single instance of the **RTOMAS** organization—i.e some IMPROVER agents lead by a single ACTUATOR, and the necessary artifacts— will be enough. All the system actions will be oriented to optimize the process as a whole.

Considering the inherent complexity associated to trying to optimize the plant as a big single process, several instances of **RTOMAS** organization could be deployed, each trying to optimize a particular subsystem. In that configuration there will be several ACTUATOR agents, each of them acting over an specific subprocess and been filled with candidate solutions from IMPROVER instances, which have been focused in the same subprocess. Agents of the same breed will have different names, in order to be able to identify each of them.

In the former scenario, some kind of global evaluation could be done, similar to the overall system structure proposed by Darby and White [10] to verify functional uniformity and common economic and operational organization's objectives. For that sake, a new

breed of agents could be introduced in the ecosystem as part of the OPTIMIZER architectural component, doing a real-time evaluation of the intended actions of ACTUATOR instances. New agents-artifacts relations should be defined as well.

Another approach for distributed optimization could be to have IMPROVER agents communicating and collaborating as a coalition, intending to send solutions with proved value for the global plant optimum. In this case, no direct supervision should be needed, the global optimization will be an indirect result of the collaboration of several IMPROVERS.

8.4 RTO with software agents

The joint of MAS and RTO fields, can produce the synthesis of new concepts and techniques for both fields, bringing benefits totally unimagined. Several results from the MAS research can find a spot for application inside the study of RTO methods.

The *agents* approach creates a good substratum to study in practice unexplored combinations of RTO techniques and test them in real production units. The use of properties from artifacts, actions from agents and configuration parameters from both, together with the power of an agents programming language like Jason, creates a fertile soil where new features can be created and fresh concepts introduced. All that could lead to new ideas and theories about what else can be done in order to increase the process's performance or improve some of its properties.

8.5 RTO approaches in a high level language

One of the main differences between RTOMAS and other RTO systems is how and where scripts for RTO approaches are located or find a body. Implementing an RTO system as a MAS the way it was done in this work, means that scripts will reside in the mindset of a software agent.

A direct implication of the former paragraph is the use of a high level language—e.g. Jason— for writing the script. That facilitates a lot the translation of an RTO approach into lines of software code, as the language to use will be nearer to the human language. As consequence, the understanding, analysis and maintenance of the script gets easier when compared to doing the same in a programming language like C++ or Java.

As it can be checked in the prototype source code, the most part of the structures built with Jason are of generic functionality, meaning they can be applied to which ever real case. As general action plans were defined for each agent breed, when a new application of

RTOMAS is faced, just some elements of **RTOMAS** need to be redefined or modified: plant models, optimization problems and metadata. That facilitates a lot the adaptation of the system to new scenarios.

8.6 RTO as an open system

Implementing RTO by means of a MAS, turns it into an open system. New agents developed by other entities can join it and use its services while providing new ones, since they comply with the organization rules and know how to use the environmental artifacts. This brings direct benefits regarding the system's maintenance and evolution, as those actions can take place without the need to shutdown the whole system. Malfunctioning agents can be replaced by new ones, and just when they are operative old instances can be stopped. Same thing for agents bringing fresh functionalities to the ecosystem. That way system operation will not be interrupted, making it more real-time.

The definition of new roles inside the agents organization is also a possibility, since it is well justified and makes totally sense. If at some point, it stays clear the need of a new breed of agents or a new artifact type, it is perfectly possible to enrich the organization definition.

Another derivation from implementing RTO as open systems, in which agent interactions occur, brings a natural phenomena of the *Complex Systems* field into the RTO one: the possibility of emergent behaviors [182]. An *emergency*, in terms of a *complex system*, is the result of the interactions of interdependent components. The behavior of a complex system cannot be inferred from the independent behavior of its several components. New structures, properties and interaction patterns can emerge during the system existence. In terms of RTO it means that new insights can be extracted from the observation of long term system runs, where several RTO approaches and techniques coexist. That is an aspect that can differentiate **RTOMAS** from other commercial products, in which there is no space for emergent behaviors, as all tasks are very well scripted.

In summary, the proposition behind **RTOMAS** brings a plethora of possibilities and interesting options for academic uses. The system prototype opens opportunities for the exploration of RTO issues. It enables deeper researches, handles distributed optimization and open new horizons for more sophisticated RTO applications, emphasizing its role as a workbench for academic research.

9 CONCLUSIONS

- A software architecture was designed as internal organization for RTO programs considering their reactive nature. Functional and quality attributes requirement were considered during the design.
- The benefits of the designed software architecture were analyzed.
- By implementing an RTO application prototype, it was proved the feasibility to build a RTO system as a MAS.
- The RTO application prototype was adapted to a real case—i.e., the optimization of a distillation unit of REPLAN refinery—, proving its applicability and usability.
- The benefits of the developed prototype for adapting it to other RTO implementations were analyzed.
- The designed architecture and the MAS approach introduced a novel internal organization for an RTO application that increases the visibility of the RTO concerns and facilitates their handling, stimulating and expediting research actions on the field.

10 IDEAS FOR FUTURE WORK

- To use the concepts and tools provided by the MOISE framework [166] to implement a better formalization of **RTOMAS** as an organization of agents.
- To develop custom GUIs for **RTOMAS**' artifacts and agents, in order to increase the usability of the system and provide further services.
- To test the **RTOMAS** prototype with a real plant control interface like the provided by the PI system [41].
- To implement other RTO approaches in the **RTOMAS**'s way: programing new agents.
- To implement a distributed optimization with the **RTOMAS** prototype.

REFERENCES

- [1] Krishnan S. et al. "Robust Parameter Estimation in On-Line Optimization - Part I. Methodology and Simulated Case Study". In: *Computers Chem. Eng.* 16.6 (1992), pp. 545–562.
- [2] L. A Youle P. B.; Duncanso. "On-line Control of Olefin Plant". In: *Chem. Proc. Eng.* 51.5 (1970), p. 49.
- [3] Chachuat B. et al. "Adaptation strategies for real-time optimization". In: *Computers Chem. Eng.* 33 (2009), pp. 1557–1567.
- [4] R Braunschweig B.; Gani. "Software Architectures and Tools for Computer Aided Process Engineering". In: *Computer Aided Chemical Engineering* 11.86 (2002), pp. 1–700.
- [5] Wooldridge M. *An Introduction to Multi-Agent Systems. Second Edition.* John Wiley & Sons, 2009. ISBN: ISBN-10: 0470519460, ISBN-13: 978-0470519462.
- [6] Pnueli A. "Specification and development of reactive systems". In: *Information Processing, Elsevier Science, Amsterdam* 86 (1986), 845858.
- [7] B. Chen C. Y.; Joseph. "On-line optimization using a two-phase approach". In: *Ind. Eng. Chem. Res.* 26.9 (1987), pp. 1924–1930.
- [8] X. Chen. "The Optimal Implementation of On-line Optimization for Chemical and Refinery Processes". Ph.D Disertation Thesis. Louisiana State University, Agriculture, and Mechanical College, 1998. Chap. 1.
- [9] R. T. Cutler C. R.; Perry. "Real time optimization with multivariable control is required to maximize profits". In: *Comp. Chem. Eng.* 7.5 (1983), pp. 663–667.
- [10] D. C. Darby M. L.; White. "On-Line Optimization of Complex Process Units". In: *Chem. Eng. Prog.* 67.10 (1988), pp. 51–59.
- [11] M. N. Rafique. "Real Time Optimization of Chemical Processes". M.Sc. Dissertation Thesis. Curtin University of Technology, 2009.
- [12] Darby M. L. et al. "RTO: An overview and assessment of current practice". In: *Journal of Process Control* 21 (2011), pp. 874–884.
- [13] W. Kadam J. V.; Marquardt. "Integration of economical optimization and control for intentionally transient process operation". In: *Assessment and Future Directions of Nonlinear Model Predictive Control International workshop, Berlin-Heidelberg, Springer* 358 (2007), pp. 419–434.

- [14] M. Garcia C. E.; Morari. "Optimal operation of integrated processing systems. Part I: open-loop on-line optimizing control". In: *AIChE Journal* 27.6 (1981), pp. 960–968.
- [15] P. L. Naysmith M. R.; Douglas. "Review of Real Time Optimization in the Chemical Process Industries". In: *Dev. Chem. Eng. Mineral Process* 3 (1995), pp. 67–87.
- [16] S. M. S. Neiro. "Integrated planning of production-distribution in petroleum supply chains". Ph. D. Dissertation Thesis. São Paulo, Brazil: University of São Paulo, 2004.
- [17] L. H. Jose R. A.; Ungar. "Pricing interprocess streams using slack auctions". In: *AIChE journal* 46.3 (2000), pp. 575–587.
- [18] Bailey J. K. et al. "Nonlinear Optimization of a Hydrocracker Fractionation Plant". In: *Computers and Chemical Engineering* 17 (1993), pp. 123–138.
- [19] G. Arkun Y.; Stephanopoulos. "Optimizing Control of Industrial Chemical Processes: State of Art Review". In: *Joint Automatic Control Conference* (1980).
- [20] George E.P. Box. "Evolutionary Operation: A Method for Increasing Industrial Productivity". In: *Journal of the Royal Statistical Society* 6.2 (1957), pp. 81–101.
- [21] A. Xiong Q.; Jutan. "Continuous optimization using dynamic simplex method". In: *Chemical Engineering Science* 58 (2003), pp. 3817–3828.
- [22] Ellis J. E. et al. "Rationalization and application of algorithms for integrated system optimization and parameter estimation". In: *International Journal of Systems Science* 24.2 (1993), pp. 219–229.
- [23] Forbes J. F. et al. "Model adequacy requirements for optimizing plant operation". In: *Computers & Chem. Eng.* 18.6 (1994), pp. 497–510.
- [24] Jang S. et al. "On-line Optimization of Constrained Multivariable Chemical Processes". In: *AIChE Journal* 33.1 (1987).
- [25] J.G. Pantelides C. C.; Renfro. "The online use of first-principles models in process operations: Review, current status and future needs". In: *Computers and Chemical Engineering* 51.5 (2013), pp. 136–148.
- [26] Forbes J. F. et al. "Model Accuracy Requirements for Economic Optimizing Model Predictive Controllers-The Linear Programming Case". In: *Proceedings of the American Control Conference. American Automatic Control Council, Green Valley, Arizona* 2 (1992), pp. 1587–1593.
- [27] Mendoza D. F. et al. "Assessing the Reliability of Different Real-Time Optimization Methodologies". In: (2015).

- [28] T. E. Miletic I. P.; Marlin. "On-line Statistical Results Analysis in Real-Time Operations Optimization". In: *Ind. Eng. Chem. Res.* 37 (1998), pp. 3670–3684.
- [29] Object Management Group (OMG). *Unified Model Language (UML) Specification, Version 2.0*. OMG Document Number formal/2005-01-01. 2005. URL: <http://www.omg.org/spec/UML/2.0>.
- [30] Marchetti A. et al. "Modifier-Adaptation Methodology for Real-Time Optimization". In: *Ind. Eng. Chem.* 48 (2009), pp. 6022–6033.
- [31] P. D. Roberts. "An algorithm for steady-state system optimization and parameter estimation". In: *International Journal of Systems Science* 10.7 (1979), pp. 719–734.
- [32] Lubansky A. S. et al. "A general method of computing the derivative of experimental data". In: *AIChE Journal* 52.1 (2006), pp. 323–332.
- [33] P. Brdys M.; Tatjewski. "An algorithm for steady-state optimizing dual control of uncertain plants". In: *IFAC Workshop on New Trends in Design of Control Systems, Smolenice, Slovak Republic* (1994).
- [34] Becerra V. M. et al. "Novel developments in process optimization using predictive control". In: *Journal of Process Control* 8.2 (1998), pp. 117–138.
- [35] J.E. Mansour M.; Ellis. "Comparison of methods for estimating real process derivatives in on-line optimisation". In: *Appl. Math. Model.* 27 (2003), pp. 275–291.
- [36] T. W. C. Roberts P. D.; Williams. "On an algorithm for combined system optimization and parameter estimation". In: *Automatica* 17 (1981), pp. 199–209.
- [37] A. W. Kuhn H. W.; Tucker. "Nonlinear programming". In: *Proceedings of 2nd Berkeley Symposium. Berkeley: University of California Press* 6.2 (1951), 481492.
- [38] Bunin G. A. et al. "Sufficient Conditions for Feasibility and Optimality of Real-Time Optimization Schemes-I. Theoretical Foundations." In: *ARXIV* (2013). URL: <http://arxiv.org/abs/1308.2620>.
- [39] Bunin G. A. et al. "Sufficient Conditions for Feasibility and Optimality of Real-Time Optimization Schemes-II. Implementation Issues." In: *ARXIV* (2013). URL: <http://arxiv.org/abs/1308.2625>.
- [40] Cubillos F. A. et al. "Real-Time Process Optimization based on Grey-Box Neural Models". In: *Brazilian Journal of Chemical Engineering* 24.3 (2007), pp. 433–443.
- [41] *PI System*. OSIssoft. URL: <https://www.osissoft.com/pi-system/>.
- [42] R. R. Rhinehart. "Automated Steady and Transient State Identification in Noisy Processes". In: *American Control Conference, Washington, DC, US* (2013), pp. 4477–4493.

- [43] Rhinehart Gore. "Evaluation of Automated Steady State and Transient State Detection Algorithms". In: (2014).
- [44] Korbel M. et al. "Steady state identification for on-line data reconciliation based on wavelet transform and filtering". In: *Computers and Chemical Engineering* 63 (2014), pp. 206–218.
- [45] R. R. Bethea R. M.; Rhinehart. *Applied Engineering Statistics*. Ed. by Marcel Dekker. New York, US, 1991.
- [46] R. R. Cao S.; Rhinehart. "An efficient method for on-line identification of steady state". In: *Journal of Process Control* 5.6 (1995), pp. 363–374.
- [47] G. Bakshi B. R.; Stephanopoulos. "Representation of process trends: III. Multiscale extraction of trends from process data". In: *Computers and Chemical Engineering* 18 (1993), pp. 267–302.
- [48] Flehmig F. et al. "Identification of trends in process measurements using the wavelet transform". In: *Computers and Chemical Engineering* 22 (1998), pp. 491–496.
- [49] Jiang T. et al. "Application of a steady state detection method based on wavelet transform". In: *Computers and Chemical Engineering* 27 (2003), pp. 569–578.
- [50] Jiang T. et al. "Industrial application of wavelet transform to the on-line prediction of side draw qualities of crude unit". In: *Computers and Chemical Engineering* 24 (2000), pp. 507–512.
- [51] Le Roux G. A. C. et al. "Improving Steady-State Identification". In: *Computer Aided Chemical Engineering* 25 (2008), pp. 459–464.
- [52] A. Bendat J.; Piersol. *Random data: analysis and measurements procedures*. Ed. by Marcel Dekker. John Wiley & Sons, 2000.
- [53] A; Golay M. J. E. Savitzky. "Smoothing and Differentiation of Data by Simplified Least Squares Procedures". In: *Anal. Chem.* 36 (1964), pp. 162–163.
- [54] V Vasyutynskyy. "Passive monitoring of control loops in building automation". In: *Technical Report, Technische Universitat Dresden, Department of Computer Science* (2005).
- [55] S. L. Alekman. "Significance tests can determine steady-state with confidence". In: *Control for the Process Industries, Putman Publications, Chicago* 3.11 (1994), pp. 62–64.
- [56] B. Schladt M.; Hu. "Soft sensors on nonlinear steady-state data reconciliation in the process industry". In: *Chemical Engineering and Processing* 46 (2007), pp. 1107–1115.

- [57] D. R. Mann H. B.; Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other". In: *Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60.
- [58] H.B. Mann. "Non-parametric tests against trend". In: *Econometrica* 13 (1945), pp. 163–171.
- [59] M.G. Kendall. *Rank Correlation Methods*. London, UK: Charles Griffin, 1975.
- [60] Kim M. et al. "Design of a steady-state detector for fault detection and diagnosis of a residential air conditioner". In: *International Journal of Refrigeration* 31 (2008), pp. 790–799.
- [61] P. C. Mahalanobis. "On the generalised distance in statistics". In: *Proceedings of the National Institute of Sciences of India* 2.1 (1936), 4955.
- [62] Yao Y. et al. "Batch-to-Batch steady state identification based on variable correlation and Mahalanobis distance". In: *Industrial Engineering Chemistry Research* 48.24 (2009), pp. 11060–11070.
- [63] W. Flehmig F.; Marquardt. "Detection of multivariable trends in measured process quantities". In: *Journal of Process Control* 16 (2006), pp. 947–957.
- [64] G. Jubien G.; Bihary. "Variation in standard deviation is best measure of steady state". In: *Control for the Process Industries, Putman Publications, Chicago, IL* 3.11 (1994), p. 64.
- [65] C. Svensson. "Automated Steady State Identification". In: *LinkedIn Automation & Control Engineering discussion group* (2012).
- [66] R. S. H. Mah. *Chemical Process Structures and Information Flows*. Buttenvorths, Boston, 1990.
- [67] L. T. Tjoa I. B.; Beigler. "Simultaneous Strategies for Data Reconciliation and Gross Error Detection of Nonlinear Systems". In: *Comput. Chem. Eng.* 15.10 (1991), pp. 679–690.
- [68] Liebman M. J. et al. "Efficient Reconciliation and Estimation for Dynamic Processes Using Nonlinear Programming Techniques". In: *Comput. Chem. Eng.* 16 (1992), pp. 963–986.
- [69] R. S. H. Tamhane A. C.; Mah. "Data Reconciliation and Gross Error Detection in Chemical Process Networks". In: *Technometrics* 27 (1985), pp. 409–422.
- [70] R. E. Reilly P. M.; Carpani. "Application of Statistical Theory of Adjustment to Material Balances". In: *13th Can. Chem. Eng. Cong., Montreal, Que* (1963).
- [71] B. R. Nounou M.; Bakshi. "On-line multiscale filtering of random and gross errors without process models". In: *AIChE Journal* 45.5 (1999), pp. 1041–1058.

- [72] C. M. Crowe. "Tests of Maximum Power for Detection of Gross Errors in Process Constraints". In: *AIChE Journal* 35 (1989), pp. 869–872.
- [73] C. M. Crowe. "The Maximum-power Test for Gross Errors in the Original Constraints in Data Reconciliation". In: *Canadian Journal. Chem. Eng.* 70 (1992), pp. 1030–1036.
- [74] R. S. H. Narasimhan S.; Mah. "Generalized Likelihood Ratio Method for Gross Error Identification". In: *AIChE Journal* 33 (1987), pp. 1514–1521.
- [75] W. A. Serth R. W.; Heenan. "Gross error detection and data reconciliation in steam-metering systems". In: *AIChE Journal* 32 (1986), p. 733.
- [76] Rosenberg J. et al. "Evaluation of schemes for detecting and identifying gross errors in process data". In: *Ind. Eng. Chem. Res.* 26 (1987), p. 555.
- [77] C. M. Crowe. "Recursive Identification of Gross Errors in Linear Data Reconciliation". In: *AIChE Journal* 34 (1988), pp. 541–550.
- [78] R. S. H. Narasimhan S.; Mah. "Treatment of general steady state process models in gross error identification". In: *Comput. Chem. Eng.* 13 (1989), p. 851.
- [79] Lauks U. E. et al. "On-Line Optimization of an Ethylene Plant". In: *First European Symposium on Computer Aided Process Engineering* (1991).
- [80] D. M. Terry P. A.; Himmelblau. "Data Rectification and Gross Error Detection in a Steady-state Process via Artificial Neural Networks". In: *Ind. Eng. Chem. Res.* 32 (1993), pp. 3020–3028.
- [81] P. A. Narasimhan S.; Harikumar. "Method to Incorporate Bounds in Data Reconciliation and Gross Error Detection-I, The Bounded Data Reconciliation Problem". In: *Comput. Chem. Eng.* 17 (1993), pp. 1115–1120.
- [82] L. Walter E.; Pronzato. "Identification of Parametric Models: from Experimental Data". In: *Softcover reprint of hardcover 1st ed. 1997 ed. [S.l.]: Springer* (2010).
- [83] A. N. Marlin T. E.; Hrymak. "Real-Time operations optimization of continuous processes". In: *Fifth International Conference on Chemical Process Control, CACHE/AICHE* (1997).
- [84] J. D. Perkins. "Plant wide optimization-opportunities and challenges". In: *FOCAPO, CACHE/AICHE* (1998).
- [85] Correa J. M. et al. "Sensitivity analysis of the modeling parameters used in simulation of proton exchange membrane fuel cells". In: *IEEE Trans. Energy Conv.* 20 (2005), pp. 211–218.
- [86] Hedengren L. D. et al. "Moving horizon estimation and control for an industrial gas phase polymerization reactor". In: *Proc. of American control conference (ACC), New York, NY* (2007).

- [87] R.E. Kalman. “Contributions to the theory of optimal control”. In: *Bol. Soc. Mat. Mexicana* (1960), pp. 102–119.
- [88] J B. Haseltine E. L.; Rawlings. “Critical evaluation of extended Kalman filtering and moving horizon estimation”. In: *Industrial and Engineering Chemistry Research* 44 (2005), pp. 2451–2460.
- [89] L.T. Arora N.; Biegler. “Redescending estimators for data reconciliation and parameter estimation”. In: *Computers and Chemical Engineering* 25 (2001), pp. 1585–1599.
- [90] C. S. Macdonald R. J.; Howat. “Data reconciliation and parameter estimation in plant performance analysis”. In: *AIChE Journal* 34.1 (1988), pp. 1–8.
- [91] I. B. F. Tjoa. “Simultaneous Solution and Optimization Strategies for Data Analysis”. Ph.D. Thesis. location: Carnegie Mellon University, 1991.
- [92] R. W. Özyurt D. B.; Pike. “Theory and practice of simultaneous data reconciliation and gross error detection for chemical processes”. In: *Computers & Chemical Engineering* 28.3 (2004), pp. 381–402.
- [93] F. R. Hampel. “The influence curve and its role in robust estimation”. In: *Journal of the American Statistical Association* 69.346 (1974), p. 383.
- [94] P. J. Huber. *Robust Statistics*. New York, US: Wiley, 1981.
- [95] Hoaglin D. C. et al. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1983. ISBN: 0-201-11371-6.
- [96] W. J. J. Rey. *Introduction to Robust and Quasi-Robust Statistical Methods*. Ed. by Springer. Berlin, New York, 1988.
- [97] M. H. Wright. “Direct search methods, once scorned, now respectable numerical analysis”. In: *In Proceedings of the Dundee biennial conference in numerical analysis* (1995).
- [98] D. M. Edgar T. F.; Himmelblau. *Optimization of chemical processes*. Ed. by Marcel Dekker. New York: McGraw-Hill, 2001.
- [99] B. C. Agrawal S. K.; Fabien. *Optimization of dynamic systems*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2007.
- [100] Sequeira S. E. et al. “On-line process optimization: parameter tuning for the real time evolution (RTE) approach”. In: *Computers & Chemical Engineering* 28.5 (2004), pp. 661–672.
- [101] Ferrer-Nadal S. et al. “An integrated framework for on-line supervised optimization”. In: *Comput. Chem. Eng.* 31 (2007), pp. 401–409.

- [102] R. T. Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. Doctoral. Irvine, US: University of California.
- [103] Bass L. et al. *Software Architecture in Practice. Third Edition*. Westford, Massachusetts, US: Addison-Wesley, 2013.
- [104] M. Garlan R.; Shaw. “An introduction to software architecture. Advances in Software Engineering & Knowledge Engineering, World Scientific Pub Co.” In: 2 (1993). Ed. by Ambriola & Tortola, pp. 1–39.
- [105] et.al Shaw M. “Abstractions for software architecture and tools to support them”. In: *IEEE Transactions on Software Engineering* 21.4 (1995), 314335.
- [106] D. Shaw M.; Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [107] P. Shaw M.; Clements. “A field guide to boxology: Preliminary classification of architectural styles for software systems”. In: *In Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC-97), Washington, D.C., Aug. 1997* (1997), pp. 6–13.
- [108] A. L. Perry D. E.; Wolf. “Foundations for the study of software architecture”. In: *ACM SIGSOFT Software Engineering Notes* 17.4 (1992), pp. 40–52.
- [109] C. et.al Ghezzi. *Fundamentals of Software Engineering*. Ed. by Prentice-Hall. 1991.
- [110] G. Andrews. “Paradigms for process interaction in distributed programs”. In: *ACM Computing Surveys* 1.23 (1991), 4990.
- [111] R. N. Taylor. “A component- and message-based architectural style for GUI software”. In: *IEEE Transactions on Software Engineering* 6.22 (1996), 390406.
- [112] M. V. Tanenbaum A. S.; Steen. *Distributed Systems. Principles and Paradigms*. Upper Saddle River, NJ, US: Pearson Prentice Hall, 2007.
- [113] P. Kruchten. “An ontology of architectural design decisions”. In: *Proceedings of the 2nd Workshop on Software Variability Management, Groningenm, NL Dec* (2004), pp. 3–4.
- [114] P. Pena. *Problem Seeking: An architectural programming primer*. AIA Press, 1987.
- [115] M. Shaw. “Toward higher-level abstractions for software systems”. In: *Data & Knowledge Engineering* 5 (1990), 119128.
- [116] A. Umar. *Object-Oriented Client/Server Internet Environments*. Prentice Hall, 1997.
- [117] A. Fuggetta; G. P. Picco; G. Vigna. “Understanding code mobility”. In: *IEEE Transactions on Software Engineering* 5.24 (1998), 342361.

- [118] R. S. Chin; S. T. Chanson. “Distributed object-based programming systems”. In: *ACM Computing Surveys* 1.23 (1991), 91124.
- [119] Buschmann F. et al. *A system of patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.
- [120] N. L. Kerth; W. Cunningham. “Using patterns to improve our architectural vision”. In: *IEEE Software* 1.14 (1997), pp. 53–59.
- [121] Gamma E. et al. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
- [122] R. T. et. al Monroe. “Architectural Styles, Design Patterns, and Objects”. In: *IEEE Software* 14.1 (1997), pp. 43–52.
- [123] J. O. Coplien. “Idioms and Patterns as Architectural Literature”. In: *IEEE Software* 1.14 (1997), pp. 36–42.
- [124] A.R. Soares R.D.P.; Sechhi. “EMSO: A new Environment for Modeling, Simulation and Optimization”. In: *Comput. Aided Chem. Eng.* 14 (2003), pp. 947–952.
- [125] Graesser Franklin. “Is it an Agent, or Just a Program- A Taxonomy for Autonomous Agents”. In: (1996).
- [126] M. Wooldridge R.H. Bordini J.F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason*. 2007.
- [127] S.P. Genesereth M.R.; Ketchpel. “Software Agents”. In: *Communications of the ACM* 37.7 (1994), pp. 48–53.
- [128] N.R. Wooldridge M.J.; Jennings. “Intelligent Agents: Theory and Practice”. In: *Knowledge Engineering Review* 10.2 (1995), pp. 115–152.
- [129] P. Russell S.J.; Norvig. *Artificial Intelligence: a Modern Approach. 2nd ed.* Prentice Hall, 2003.
- [130] G. A. Tosić P.T.; Agha. “Towards a hierarchical taxonomy of autonomous agents”. In: *Proceedings of IEEE International Conference on Systems, Man and Cybernetics, The Hague, Netherlands* (2004), pp. 3421–3426.
- [131] J. C. Brustoloni. “Autonomous Agents: Characterization and Requirements”. In: *Technical Report CMU-CS-91-204, Carnegie Mellon University* (1991).
- [132] F. et al. Bellifemine. *Developing Multi-Agent Systems with JADE*. West Sussex, England: John Wiley & Sons, 2007.
- [133] M. Rao A. S.; Georgeff. “Agents. from Theory to Practice”. In: *In Proceedings of the First International Conference on Multi-Agent Systems, San Francisco, CA* (1995), pp. 312–319.

- [134] Y Shoham. “Agent-oriented programming”. In: *Artificial Intelligence* 60 (1 1993), pp. 51–92.
- [135] Y. Shoham. “AGENT0: a simple agent language and its interpreter”. In: *AAAI’91 Proceedings of the ninth National conference on Artificial intelligence* 2 (1991), pp. 704–709.
- [136] A. S. Rao. “AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language”. In: *Proceedings of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)* (1996).
- [137] Atalla F. Sayda. “Multi-agent Systems for Industrial Applications: Design, Development, and Challenges”. In: *Multi-Agent Systems - Modeling, Control, Programming, Simulations and Applications*. Ed. by Faisal Alkhateeb. InTech, 2011. Chap. 23, pp. 469–494.
- [138] N. R. Jennings. “On agent-based software engineering”. In: *Artificial Intelligence* (2000), pp. 277–296.
- [139] N. Jennings. “The Archon System and its Applications”. In: *Proceedings of the 2nd International Working Conference on Cooperating Knowledge Based Systems (CKBS-94), Dake Centre, University of Keele, UK* (1994), pp. 13–29.
- [140] M. et al. Albert. “Multi-agent Systems for Industrial Diagnostics”. In: *Proceedings of 5th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes, Washington, DC* (2003), pp. 483–488.
- [141] H. Parunak. “Manufacturing Experience with the Contract Net”. In: *In Huhns, M. (ed.), Distributed Artificial Intelligence, Pitman, London* (1987), pp. 285–310.
- [142] N. et al. Neagu. “LS/ATN: Reporting on a Successful Agent-Based Solution for Transport Logistics Optimization”. In: *Proceedings of the IEEE Workshop on Distributed Intelligent Systems (WDIS06), Prague* (2006).
- [143] D. et al. Greenwood. “Service Level Agreement Management with Adaptive Coordination”. In: *Proceedings of the International Conference on Networking and Services (ICNS06), Silicon Valley, USA* (2006).
- [144] *FIPA standards*. FIPA. URL: <http://www.fipa.org>.
- [145] C. A. R. Hoare. “Communicating sequential processes”. In: *Communications of the ACM* 21 (8 1978), pp. 666–677.
- [146] R. Milner. *Communication and concurrency*. Ed. by Inc. Upper Saddle River Prentice-Hall. NJ, USA, 1989. ISBN: 0-13-115007-3.
- [147] Finin T. et al. “KQML as an agent communication language”. In: *CIKM 94. Proceedings of the third international conference on Information and knowledge management* (1994), pp. 456–463.

- [148] F. Grijó. “Multi-agent system organization: An engineering perspective”. In: *In Pre-Proceeding of the 10th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW2001)* (2001).
- [149] L. Gasser. “Organizations in multi-agent systems”. In: *In Pre-Proceeding of the 10th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW2001)* (2001).
- [150] Fox M. S. et al. “An organizational ontology for enterprise modeling”. In: *Simulating Organizations: Computational Models of Institutions and Groups*. Menlo Park CA: AAAI/MIT Press (1998), pp. 131–152.
- [151] M. Weber. *Economy and Society*. Ed. by University of California Press. Berkeley, CA, 1987.
- [152] T. W. MALONE. “Tools for inventing organizations: Toward a handbook of organizational process”. In: *Management Science* 45.3 (1999), 425443.
- [153] P. K. Manning. “Rules in an Organizational Context”. In: *Organizational Analysis*, Sage (1977).
- [154] March J. G et al. *Dynamics Of Rules: Change In Written Organizational Codes*. Stanford, CA: Stanford University Press, 2000.
- [155] Jaime S. Sichman. “MAS Organizations”. In: *Class room lecture* (2007).
- [156] P. Bernoux. “La sociologie des organisations”. In: *Seuil, Paris* (1985).
- [157] C. B. Lemaître C.; Excelente. “Multi-agent organization approach”. In: *Proceedings of II Iberoamerican Workshop on DAI and MAS, Toledo, Spain* (1998).
- [158] G. et al. Picard. “Reorganization and self-organization in Multi-Agent Systems”. In: (2010).
- [159] D. Goldberg A.; Robson. *Understanding Robust and Exploratory Data Analysis*. New York, US: Wiley, 1983.
- [160] O. Dahl. “From Object-Orientation to Formal Methods, Cap. The Birth of Object-Orientation: The Simula Languages”. In: *Lecture Notes in Computer Science*, Springer-Verlag Berlin Heidelberg 2635 (2004).
- [161] G. Booch. *Object-Oriented Analysis and Design*. California, US: Addison Wesley, 1994.
- [162] D. M. Kernighan B. W.; Ritchie. *The C Programming Language*. Ed. by Prentice Hall Professional Technical Reference. 1988. ISBN: 0131103709.
- [163] B. Stroustrup. *The C Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201700735.
- [164] Bossier et al. “Multi-Agent Oriented Programming with JaCaMo”. In: (2011).

- [165] A. et al. Ricci. “Environment programming in multi-agent systems: an artifact-based perspective, Autonomous Agents and Multi-Agent Systems”. In: 23 (2011), 158192.
- [166] Hübner J. F. et al. “Developing organized multi-agent systems using the MOISE+ model: programming issues at the system and agent levels”. In: *Agent-Oriented Software Engineering* 1.3-4 (2007), pp. 370–395.
- [167] Omicini A. et al. “Artifacts in the A&A meta-model for multi-agent systems”. In: *Autonomous Agents and Multi-Agent Systems* 17.3 (2008), pp. 432–456.
- [168] Weyns D. et al. “Environment as a first-class abstraction in multi-agent systems, Autonomous Agents and Multi-Agent Systems”. In: 14.1 (2007), pp. 5–30.
- [169] Hübner F. et al. “Instrumenting multi-agent organisations with organisational artifacts and agents”. In: *Autonomous Agents and Multi-Agent Systems* 20 (2010), pp. 369–400.
- [170] R. E. Williams T. J.; Otto. “A Generalized Chemical Processing Model for the Investigation of Computer Control”. In: *American Institute of Electrical Engineers, Part I: Communication and Electronics. Transactions* 79.5 (1960), pp. 458–473.
- [171] C. Smith B.; Welty. “Ontology: Towards a New Synthesis”. In: *Introduction to the Second International Conference on Formal Ontology and Information Systems, Ogunquit, Maine, USA* (2001).
- [172] PostgreSQL Global Development Group. *PostgreSQL*. 2017. URL: <https://www.postgresql.org/>.
- [173] *Extensible Markup Language*. URL: <https://www.w3.org/XML/>.
- [174] *MATLAB*. MathWorks. URL: <https://www.mathworks.com/products/matlab.html>.
- [175] *Pentaho Data Integration tool*. PENTAHO. URL: <http://www.community.pentaho.com/projects/data-integration/>.
- [176] *PostgreSQL Database Server*. URL: <http://www.postgresql.org>.
- [177] L Wachter A.; Biegler. “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming”. In: *Mathematical Programming* 106.1 (2006), pp. 25–57.
- [178] J. E. A. Graciano. “Real Time Optimization in chemical processes: Evaluation of strategies, improvements and industrial application”. PhD. Dissertation Thesis. University of São Paulo, 2015.
- [179] D.F.M et al. Muoz. “Real-time Optimization of an Industrial-Scale Vapor Recompression Distillation Process. Model Validation and Analysis.” In: *Ind. Eng. Chem. Res* 52 (2013), pp. 5735–5746.

- [180] V. H. Fariss R. H.; Law. “An efficient computational technique for generalized application of maximum likelihood to improve correlation of experimental data.” In: *Computers and Chemical Engineering* 3 (1979), pp. 95–104.
- [181] *Optimization with or without restriction using the flexible polyhedral method*. URL: <http://www.enq.ufrgs.br/enqlib/numeric/complex.zip>.
- [182] Y. Bar-Yam. *Dynamics of complex systems*. US: Addison-Wesley, 1997. ISBN: 0-201-55748-7.

Appendix A – Steady-state model for the Williams-Otto reactor

```

using "types";

Model W0reactor

PARAMETERS
  f1 as Real (Brief="Reparametrized frequency coefficient of first
    reaction" ,Default=1);
  f2 as Real (Brief="Reparametrized frequency coefficient of second
    reaction",Default=1);
  f3 as Real (Brief="Reparametrized frequency coefficient of thrid
    reaction" ,Default=1);
  Ea1 as Real (Brief="Reparametrized Activation energy of first reaction
    " ,Default=1);
  Ea2 as Real (Brief="Reparametrized Activation energy of second
    reaction",Default=1);
  Ea3 as Real (Brief="Reparametrized Activation energy of thrid reaction
    " ,Default=1);
  Mt as Real (Brief="Total reactor mass");
  Tr as Real (Brief="Reactor temperature in Celcius degrees", Default
    =90, Lower=80, Upper=100);
  Fb as Real (Brief="Feed flow rate of B", Default=4, Lower=4, Upper
    =6);

VARIABLES

  Fa as Real (Brief="Feed flow rate of A", Default=2, Lower=0, Upper=4);
  Fr as Real (Brief="Outlet reactor flowrate", Lower=2, Upper=10);
  r1 as Real (Brief="First reaction rate", Lower=0, Upper=5);
  r2 as Real (Brief="Second reaction rate", Lower=0, Upper=5);
  r3 as Real (Brief="Third reaction rate", Lower=0, Upper=5);
  Xa as Real (Brief="Mass fraction of component A", Default=0.2, Lower
    =0, Upper=1);
  Xb as Real (Brief="Mass fraction of component B", Default=0.2, Lower
    =0, Upper=1);
  Xc as Real (Brief="Mass fraction of component C", Default=0.2, Lower
    =0, Upper=1);
  Xe as Real (Brief="Mass fraction of component E", Default=0.2, Lower
    =0, Upper=1);
  Xp as Real (Brief="Mass fraction of component P", Default=0.2, Lower
    =0, Upper=1);

```

```

Xg as Real (Brief="Mass fraction of component G", Default=0.2, Lower
=0, Upper=1);
K1 as Real (Brief="Kinetic constant of first reaction", Lower=0,
Upper=5);
K2 as Real (Brief="Kinetic constant of second reaction", Lower=0,
Upper=5);
K3 as Real (Brief="Kinetic constant of third reaction", Lower=0,
Upper=5);

F0 as Real (Brief="Profit - Objective function");
p1 as Real (Brief="Penalt Function for Fb - lower");
p2 as Real (Brief="Penalt Function for Fb - upper");
p3 as Real (Brief="Penalt Function for Tr - lower");
p4 as Real (Brief="Penalt Function for Tr - upper");

Tr_v as Real (Brief="Reactor temperature in Celcius degrees", Default
=90, Lower=80, Upper=100);
Fb_v as Real (Brief="Feed flow rate of B", Default=4, Lower=4, Upper
=6);

```

EQUATIONS

```

#Kinetics constants
K1 = f1*exp(-Ea1/(Tr_v+273.15));
K2 = f2*exp(-Ea2/(Tr_v+273.15));
K3 = f3*exp(-Ea3/(Tr_v+273.15));

#Reactions rates
r1 = K1*Xa*Xb*Mt;
r2 = K2*Xb*Xc*Mt;
r3 = K3*Xc*Xp*Mt;

#Global mass balance
Fr = Fa+Fb_v;

#Component mass balance
Fr*Xa = Fa - r1;
Fr*Xb = Fb_v - r1 - r2;
Fr*Xc = + 2*r1 - 2*r2 - r3;
Fr*Xe = + 2*r2;
Fr*Xp = + r2 - 0.5*r3;
Fr*Xg = + 1.5*r3;

F0 = -1143.38*Xp*Fr -25.92*Xe*Fr +76.23*Fa +114.34*Fb_v;

#for optimization
if (Fb_v < 4) then

```

```
    1e-6*p1 = 1e-6*(4 - Fb_v)*1e4;  
else  
    p1 = 0;  
end  
  
if (Fb_v > 6) then  
    1e-6*p2 = 1e-6*(Fb_v - 6)*1e4;  
else  
    p2 = 0;  
end  
  
if (Tr_v < 80) then  
    1e-6*p3 = 1e-6*(80 - Tr_v)*1e3;  
else  
    p3 = 0;  
end  
  
if (Tr_v > 100) then  
    1e-6*p4 = 1e-6*(Tr_v - 100)*1e3;  
else  
    p4 = 0;  
end  
  
end
```

Appendix B – EMSO Flowsheet for the Williams-Otto case

```

using "Model_WO";

FlowSheet Williams_Otto

DEVICES

    CSTR as W0reactor;

SET

    CSTR.Mt = 2105.02;#[kg]
    CSTR.f1  = 1.6599e6;
    CSTR.f2  = 7.2117e8;
    CSTR.f3  = 2.6745e12;

SPECIFY

    CSTR.Fa = 1.875;#[kg/s]

    #DoF - Optimization
    CSTR.Fb_v = CSTR.Fb;#[kg/s]
    CSTR.Tr_v = CSTR.Tr;#[degC]

OPTIONS

    Dynamic = false;
    GuessFile = "Guess_WO.rlt";

    NLASolver(
        File = "nlasolver",
        RelativeAccuracy = 1e-6,
        AbsoluteAccuracy = 1e-6,
        MaxIterations    = 50
    );

end

```

Appendix C – EMSO Optimization for the Williams-Otto case

```

using "Flow_WO";

Optimization Opt_WO as Willians_Otto

MINIMIZE
  - CSTR.FO + CSTR.p1 + CSTR.p2 + CSTR.p3 + CSTR.p4;

FREE
  CSTR.Fb_v;
  CSTR.Tr_v;

GUESS
  CSTR.Fb_v=6;
  CSTR.Tr_v=80;

OPTIONS
  Dynamic      = false;
  NLPsolveNLA  = false;
  FeasiblePath= true;
  GuessFile    = "Guess_WO.rlt";

NLASolver(File="sundials", RelativeAccuracy=1e-6, AbsoluteAccuracy=1e-6,
  MaxIterations=500);

NLPSolver(
  File = "ipopt_emso",
  MaxIterations      = 500,
  RelativeAccuracy   = 1e-8,
  AbsoluteAccuracy   = 1e-8
);
end

```


Appendix D – Steady-state model for the REPLAN case

```

using "types","streams";

Model CompleteFlowsheet

PARAMETERS

outer PP as Plugin (Brief="External Physical Properties",Type="PP");
outer NComp as Integer (Brief="Number of components");
Eir1 as Real (Brief="Individual Murphree Efficiency, Rectification
trays 101 - NF", Lower = 0.2, Upper = 2.0);
Eir2 as Real (Brief="Individual Murphree Efficiency, Rectification
trays 1 - 100", Lower = 0.2, Upper = 2.0);
Eis as Real (Brief="Individual Murphree Efficiency, Stripping", Lower
= 0.2, Upper = 2.0);
Propy_p as Real (Brief="Propylene feed concentration" ,Lower = 0.2,
Upper = 0.9);
F_feed_p as Real (Brief="Feed flow rate [kg/h]", Lower = 2.5e4,
Upper = 6e4);
Lref_mass_p as Real (Brief="Reflux flow rate [kg/h]", Lower = 2.5e5,
Upper = 4.5e5);
PB_mass_p as Real (Brief="Botton flow rate [kg/h]", Lower = 1e3, Upper
= 2e4);
vfB as fraction (Brief="Vapor Fraction in Boil Up Stream", Lower = 0,
Upper = 1);
U_cooler as Real (Brief="Coeficient gobal", Lower = 200, Upper = 1000)
;
PcompOut as pressure (Brief="Compressor Outlet Pressure - C-9701");
P_top as Real (Brief="Top pressure [kg/h]", Lower = 8, Upper = 12);
P_bot as Real (Brief="Botton pressure [kg/h]", Lower = 9, Upper = 14);
QTow as Real (Brief="Column heat loss [m^2*kg/s^3]", Lower = 0, Upper
= 20000);
DeltaPump as pressure (Brief="Pump Delta Pressure - B-9708 A/B");

.....

VARIABLES

CompIn as vapour_stream (Brief="Compressor Intlet C-9701");
CompOut_mass as Real (Brief="Compressor Outlet stream mass flow");
CompOut as vapour_stream (Brief="Compressor Outlet C-9701");
D as liquid_stream (Brief="Distillate");

```

```

D_mass as Real (Brief="Distillate flow-mass base");
F_new as liquid_stream (Brief="Feed");
Fl as liquid_stream (Brief="Liquid Feed");
Fv as vapour_stream (Brief="Vapor Feed");
FBIn as stream (Brief="Reboiler/Condenser Inlet Stream-tube side");
FBIn_mass as Real(Brief="Reboiler/Condenser Outlet Stream-mass base");
FBOut as liquid_stream (Brief="Reboiler/Condenser Outlet Stream - tube
    side");
FBOut_l as liquid_stream (Brief="Reboiler/Condenser Liquid Outlet
    Stream - tube side");
FBOut_v as vapour_stream (Brief="Reboiler/Condenser Vapor Outlet
    Stream - tube side");
FV2VapourOut as vapour_stream (Brief="Outlet of Expansion Valve,
    Reboiler/Condenser Cooler");
FV2LiquidOut as liquid_stream (Brief="Outlet of Expansion Valve,
    Reboiler/Condenser Cooler");
F_mass as Real (Brief="Inlet flow - mass base");
LinB as stream (Brief="Reboiler/Condenser Inlet Stream, shell side");
LoutB as liquid_stream (Brief="Reboiler/Condenser Outlet Stream, shell
    side");
Lreflux as liquid_stream (Brief="Reflux");
Lreflux_v as vapour_stream (Brief="Vapor fraction after reflux valvule
    ");
Lreflux_l as liquid_stream (Brief="Liquid fraction after reflux
    valvule");
Lreflux_mass as Real (Brief="Reflux - mass base", Upper=4.20000 ,
    Lower=3.00000, Default =3.50000 );
Lsump as liquid_stream (Brief="Sump Liquid Outlet Stream");
Ltray(Nt) as liquid_stream (Brief="Liquid Flows Through Column");

.....

SET
Mw = PP.MolecularWeight();
DPjr = (P_bot-P_top)/Nt*'kgf/cm^2';
DPjs = (P_bot-P_top)/Nt*'kgf/cm^2';

EQUATIONS

#Input Parameters
F_mass = F_feed_p *1e-3; #[kg/h]
F_new.z(2) = Propy_p;

#Sumation concentration feed stream
sum(F_new.z)=1;

#Cost equations

```

```

Profit = - Feed_cost + D_cost + PB_cost - Comp_cost - water_cost;

Feed_cost  = F_mass*1e3*1.5485*'R$/h';
D_cost     = D_mass*1e3*3.4850*'R$/h';
PB_cost    = PB_mass*1e3*0.8924*'R$/h';
Comp_cost  = Win*1e5*303.22*'R$/(MW*h)';
water_cost = Wf*14.8*'R$/kg'/1e6*3.5; #dollar 3.5

Propa_top  = D.F*D.z(3)*Mw(3)/(D_mass*1e3)*'h/kg'*1e6;
Etan_top   = D.F*D.z(1)*Mw(1)/(D_mass*1e3)*'h/kg'*1e6;

#"DISTILLATION COLUMN: FEED CONDITION (ADIABATIC EXPANSION)"
F_new.F      = Fv.F      + Fl.F      ;
F_new.F*F_new.z = Fv.F*Fv.z + Fl.F*Fl.z;

10E-5*(F_new.F*F_new.h) = 10E-5*(Fv.F*Fv.h + Fl.F*Fl.h);
Fl.T = Fv.T;

1e3*(Fv.z*PP.VapourFugacityCoefficient(Fv.T,Fv.P,Fv.z)) = 1e3*(Fl.z*
    PP.LiquidFugacityCoefficient(Fl.T,Fl.P,Fl.z));
1e2*sum(Fl.z) = 1e2*1;

10E-5*(Fl.P) = 10E-5*(Vtray(Nf).P);
10E-5*(Fv.P) = 10E-5*(Vtray(Nf).P);

....

```

Appendix E – EMSO Flowsheet for the REPLAN case

```

using "types","streams","Model_REPLAN";

FlowSheet ColumnFlowSheet

    PARAMETERS
    PP as Plugin(Brief          = "Physical Properties",
                Type            = "PPIISE",
                Project          = "Project_IISE.ise",
                FlashTolerance = 1e-9);

    NComp as Integer(Brief="Number of Components");

    SET
    NComp = PP.NumberOfComponents;

    DEVICES
    FS as CompleteFlowsheet;

    SET
    #RTO_PARAMS_START
    FS.Eir1      = 1.2;
    FS.Eir2      = 1.2;
    FS.Eis       = 1.0;
    FS.Propy_p   = 0.772642;
    FS.F_feed_p  = 28055.0;
    FS.Lref_mass_p = 364383.0;
    FS.PB_mass_p = 6897.02;
    FS.P_top     = 9.92334;
    FS.P_bot     = 11.37;
    FS.QTow      = 3949.72;
    FS.vfB       = 0.25;
    FS.U_cooler  = 850.151;
    #RTO_PARAMS_END

    #Set parameters
    FS.PcompOut  = 16.17 *'kgf/cm^2'; #Compressor outlet pressure
    FS.DeltaPump = 2.45  *'kgf/cm^2';
    FS.Pv        = 13.45 *'kgf/cm^2'; #Tank outlet pressure
    FS.Twater_in = (26+273.15) *'K';

    #Fixed parameters

```

```

FS.Nt          = 197;
FS.Nf          = 157;
FS.PdropB      = 0.1      *'kgf/cm^2';
FS.PdropC      = 0.1      *'kgf/cm^2';
FS.Qbl         = 0        *'m^2*kg/s^3';
FS.Qcl         = 0        *'m^2*kg/s^3';
FS.Area_c      = 1686     *'m^2';
FS.delta_b_heur = 0.547   *'K';

SPECIFY
#COLUMN'S FEED
FS.F_new.T      = 298.11   *'K';
FS.F_new.P      = 11.49    *'kgf/cm^2';
FS.F_new.z(1)   = 0.000100;
FS.F_new.z(4)   = 0.000001;
FS.F_new.z(5)   = 0.000001;
FS.F_new.z(6)   = 0.000001;
FS.F_new.z(7)   = 0.000001;
FS.F_new.z(8)   = 0.000001;
FS.F_new.z(9)   = 0.000001;
FS.F_new.z(10)  = 0.000001;

FS.Lreflux_mass = FS.Lref_mass_p * 1e-5; #[kg/h]
FS.PB_mass      = FS.PB_mass_p   * 1e-3; #[kg/h]

#REBOILER/CONDENSER
FS.FBOut_v.F = 0.1 *'kmol/h'; #Specifying liquid fraction in
    the cool outlet stream (Reboiler)

OPTIONS
Dynamic = false;
GuessFile = "Guess_REPLAN.rlt";
NLASolver(File = "sundials", RelativeAccuracy = 1E-6,
    AbsoluteAccuracy = 1E-6, MaxIterations = 150);

end

```

Appendix F – EMSO Parameters Estimation for the REPLAN case

```

using "types","streams","Flow_Est_REPLAN";

Estimation Col_Est as ColumnFlowSheet

    ESTIMATE

    # PAR          START      LOWER      UPPER      UNIT
    FS.Eir2         0.7        0.5        1.200;
    FS.Eir1         0.7        0.5        1.200;
    FS.Eis          0.7        0.5        1.000;
    FS.Propy_p      0.766301   0.2        0.900;
    FS.F_feed_p     28550      2.5e4      6.0e4; #[kg/h]
    FS.Lref_mass_p  364340     2.5e5      4.5e5; #[kg/h]
    FS.PB_mass_p    6730       1e3        2.0e4; #[kg/h]
    FS.P_top        10.04      8          12.00; #[kgf/cm2]
    FS.P_bot        11.37      9          13.00; #[kgf/cm2]
    FS.QTow         800        0          5000;  #[W]
    FS.vfB          0.5        0.25       1;
    FS.U_cooler     494.09     200        1000;

    EXPERIMENTS

    # FILE          WEIGHTH    TYPE
    "Data_REPLAN.dat"  1      "fit";

    OPTIONS

    Statistics( Fit = true, Parameter = false, Prediction = false);

    GuessFile = "Guess_Est_REPLAN.rlt";

    NLPsSolver( AbsoluteAccuracy=1e-8, MaxIterations = 500,
                TolX = 1e-4, TolFun = 1e-5,
                File = "rotdisc_emso");

    Dynamic = false;

end

```

Appendix G – EMSO Experiment Data File for the REPLAN case

MEASURE	FS.Vboiler.T	FS.Lreflux_mass	FS.Ltray(153).T	FS.Ltray(119).T					
	FS.PB_mass	FS.Propa_top	FS.Ltray(69).T	FS.PB.z(2)	FS.				
	P_bot	FS.P_top	FS.Ltray(137).T	FS.F_new.z(3)	FS.				
	Twater_out	FS.Ltray(17).T	FS.FBIn_mass	FS.F_new.z(2)	FS.				
	D_mass	FS.Ltray(171).T	FS.Ltray(35).T	FS.F_mass	FS.				
	FCOut_mass	FS.Ltray(85).T	FS.Ltray(51).T						
STDDEV	0.11	0.747	0.017	0.017	0.395	30.265	0.017	0.00141	
	0.1	0.1	0.017	0.00141	0.35	0.017	0.369	0.00141	0.309
		0.017	0.017	0.262	0.517	0.017	0.017		
DATA	0.0	370.0	0.0	0.0	15.034376		0.0	0.0	
	41.336918	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
		58.69706		18.114288	0.0	0.0	0.0	0.0	
	0.0	0.0							

Appendix H – EMSO Optimization for the REPLAN case

```

using "Flow_REPLAN";

Optimization Col_Opt as ColumnFlowSheet

MINIMIZE
    - FS.Profit*'h/R$'+ FS.p1 + FS.p2;

FREE
    FS.Lreflux_mass;
    FS.PB_mass;

OPTIONS
    Dynamic      = false;
    NLPsolveNLA  = false;
    FeasiblePath = true;
    GuessFile    = "Guess_REPLAN.rlt";

NLASolver(
    File = "sundials",
    RelativeAccuracy = 1e-6,
    AbsoluteAccuracy = 1e-6,
    MaxIterations    = 500
);

NLPsolver(
    File = "complex_emso",
    MaxIterations    = 500,
    RelativeAccuracy = 1e-8,
    AbsoluteAccuracy = 1e-8
);

end

```


Appendix I – MPA OPTIMIZER agent source code

```

/***** Initial beliefs and rules *****/

estimation_problem (

  file("/home/elyser/Temp/RT0/eml/Est_REPLAN.mso"),
  name("Col_Est"),
  data_file("/home/elyser/Temp/RT0/eml/Data_REPLAN.dat"),
  params(["Eir2", "Eir1", "Eis", "Propy_p", "F_feed_p", "Lref_mass_p", "
    PB_mass_p", "P_top",
    "P_bot", "QTow", "vfB", "U_cooler"])
).

optimization_problem (

  file("/home/elyser/Temp/RT0/eml/Opt_REPLAN.mso"),
  flowsheet("/home/elyser/Temp/RT0/eml/Flow_REPLAN.mso"),
  name("Col_Opt"),
  free_vars(["Lreflux_mass", "PB_mass_p"])
).

emsoc_tool (

  path("/home/elyser/Temp/RT0/eml/emsoc")
).

rto_step(ss_detection).

/***** Initial goals *****/

!process_optimized.

/***** Plans *****/

+!process_optimized : true

  <- .my_name(MyName);
    .print(MyName , " starting ...");
    !rto_step_printed;
    !observe_tags_board;
    !environment_ready;
    !register_in_ranking;

```

```

.

+!rto_step_printed

  <- ?rto_step(Step);
    .print("RTO Step: ", Step);
.

+!observe_tags_board : true

  <- ?artifact_id("TAGS_BOARD", Artifact_Id);
    focus(Artifact_Id);
    .print("Observing artifact ", "TAGS_BOARD", " (" , Artifact_Id, "
      ")
.

+!register_in_ranking : true

  <- ?artifact_id("OPTIMIZERS_RANKING", Artifact_Id);
    .my_name(MyName);
    registerAgentInRanking(MyName);
    .print("Registered in ", "OPTIMIZERS_RANKING", " (" , Artifact_Id,
      ")")
.

+?artifact_id(Artifact_Name, Artifact_Id) : true

  <- lookupArtifact(Artifact_Name, Artifact_Id)
.

-?artifact_id(Artifact_Name, Artifact_Id) : true

  <- .wait(10);
    ?artifact_id(Artifact_Name, Artifact_Id)
.

+!environment_ready : true

  <- //----- ExpDataFile -----//

    ?estimation_problem( file(_), name(_), data_file(Data_File),
      params(_));

    makeArtifact("EMSO_DataFile","br.usp.pqi.rto.art.EMSODataFile", [
      Data_File], EMSO_DataFile_Id);
    .print("Artifact ", "EMSO_DataFile", " setted up: ", " (" ,
      EMSO_DataFile_Id, ")")

```

```

//----- Optimization Flowsheet File -----//

?optimization_problem (file(_), flowsheet(Flowsheet), name(_),
    free_vars(_));

makeArtifact("EMSO_Opt_Flowsheet", "br.usp.pqi.rto.art.
    EMSO0ptFlowsheet", [Flowsheet], EMSO_Opt_Flow_Id);

.print("Artifact ", "EMSO_Opt_Flowsheet", " setted up: ", " (" ,
    EMSO_Opt_Flow_Id, ")");

//----- EMSOC -----//

?emsoc_tool ( path(EMSOC_Path));

makeArtifact("EMSOC", "br.usp.pqi.rto.art.EMSOC",[EMSOC_Path],
    EMSOC_Id);
focus(EMSOC_Id);
.print("Observing artifact ", "EMSOC", " (" , EMSOC_Id, ")")
.

+in_steady_state(Rep_State) : rto_step(ss_detection)

<- println("steady-state detected !!!");
+rto_step(params_estimation);
!rto_step_printed;

?estimation_problem( file(File), name(Problem), data_file(_),
    params(Params));

saveObservedData(Rep_State, false);
stopSampling;
solve_estimation(File, Problem, Params);
.

+estimation_solved(Estimated_Params) : true

<- .print("Model's parameters were estimated: ", Estimated_Params);

updateModelParameters(Estimated_Params);

+rto_step(optimization);
!rto_step_printed;

?optimization_problem ( file(File), flowsheet(_), name(Problem),
    free_vars(Free_Vars))

```

```
        solve_optimization(File, Problem, Free_Vars);  
.  
+optimization_solved(Free_Var_Values, Expected_Profit) : true  
  
    <- .print("New Optimization was calculated: ", Free_Var_Values);  
        .my_name(MyName);  
  
        stageOptimization(MyName, Expected_Profit, Free_Var_Values);  
  
        -+rto_step(ss_detection);  
        !rto_step_printed;  
  
        startSampling;  
.
```

Appendix J – ACTUATOR agent source code

```

/***** Initial beliefs and rules *****/

process_tags (
[
  tag(id("fs_f_feed_p"),var("FS.F_feed_p"),desc("Feed flow rate [kg/h]"),
    ,label("FS.F_feed_p")),

  tag(id("fs_pb_mass"),var("FS.PB_mass"),desc("Botton flow rate [kg/h]"),
    ,label("FS.PB_mass")),

  tag(id("fs_d_mass"),var("FS.D_mass"),desc("Distillate flow mass base"),
    ,label("FS.D_mass")),

  tag(id("fs_f_new_z_2"),var("FS.F_new.z(2)"),desc("Concent de propileno
    na entrada"),label("FS.F_new.z(2)")),

  tag(id("fs_pb_z_2"),var("FS.PB.z(2)"),desc("Concent de propileno no
    fundo"),label("FS.PB.z(2)")),

  tag(id("fs_lreflux_mass"),var("FS.Lreflux_mass"),desc(""),label("FS.
    Lreflux_mass")),

  tag(id("fs_fcout_mass"),var("FS.FCOut_mass"),desc(""),label("FS.
    FCOut_mass")),

  tag(id("fs_fbin_mass"),var("FS.FBIn_mass"),desc(""),label("FS.
    FBIn_mass")),

  tag(id("fs_f_mass"),var("FS.F_mass"),desc(""),label("FS.F_mass")),

  tag(id("fs_propa_top"),var("FS.Propa_top"),desc(""),label("FS.
    Propa_top")),

  tag(id("fs_f_new_z_3"),var("FS.F_new.z(3)"),desc(""),label("FS.F_new.z
    (3)")),

  tag(id("fs_ltray_17_t"),var("FS.Ltray(17).T"),desc(""),label("FS.Ltray
    (17).T")),

  tag(id("fs_ltray_35_t"),var("FS.Ltray(35).T"),desc(""),label("FS.Ltray
    (35).T")),

```

```

tag(id("fs_ltray_51_t"), var("FS.Ltray(51).T"), desc(""), label("FS.Ltray
(51).T")),

tag(id("fs_ltray_69_t"), var("FS.Ltray(69).T"), desc(""), label("FS.Ltray
(69).T")),

tag(id("fs_ltray_85_t"), var("FS.Ltray(85).T"), desc(""), label("FS.Ltray
(85).T")),

tag(id("fs_ltray_119_t"), var("FS.Ltray(119).T"), desc(""), label("FS.
Ltray(119).T")),

tag(id("fs_ltray_137_t"), var("FS.Ltray(137).T"), desc(""), label("FS.
Ltray(137).T")),

tag(id("fs_ltray_153_t"), var("FS.Ltray(153).T"), desc(""), label("FS.
Ltray(153).T")),

tag(id("fs_ltray_171_t"), var("FS.Ltray(171).T"), desc(""), label("FS.
Ltray(171).T")),

tag(id("fs_p_top"), var("FS.P_top"), desc(""), label("FS.P_top")),

tag(id("fs_p_bot"), var("FS.P_bot"), desc(""), label("FS.P_bot")),

tag(id("fs_twater_out"), var("FS.Twater_out"), desc(""), label("FS.
Twater_out")),

tag(id("fs_vboiler_t"), var("FS.Vboiler.T"), desc(""), label("FS.Vboiler.
T"))
]).

tags_datasource (

    url("jdbc:postgresql://localhost/rto"),
    user(rtouser),
    password(rtoadmin),
    dataset(replan_tags)
).

ss_detection_problem (

    tags(["fs_f_feed_p", "fs_pb_mass", "fs_d_mass", "fs_f_new_z_2", "
        fs_pb_z_2"]),
    time_unit(seconds),
    sampling_start_delay(2),

```

```

    sampling_freq(5),
    time_windows_size(5),
    ss_detection_start_delay(10),
    ss_detection_freq(10),
    ss_detection_method(random),
    ss_detection_use_recent_date(true)
).

instant_profit_calculation (

    class("br.usp.pqi.process.sampling.SumAllProfitFormula")
).

setpoints(
    [
        tag(id("fs_lreflux_mass"),var("Lreflux_mass")),
        tag(id("fs_pb_mass"),var("PB_mass_p"))
    ]
).

/-- OOP Poll Strategies -//

poll_strategy(first_available).
//poll_strategy(timer_since_first_available).
//poll_strategy(frequent_checking).

/-- Feasibility Checkers -//

feasibility_checkers(
    [
        checker(name("OOP_SPs_Distance_Checker"), class("br.usp.pqi.process.
            optimization.SetPointDistanceChecker"), desc(""))
    ]
).

performance_counters(
    [
        counter(name("staged_opts"), desc("Staged Optimizations count")),
        counter(name("dnf_opts"), desc("Optimizations discarded in DNF phase"))
    ]
).

/***** Initial goals *****/

!do_RT0.

```

```

/***** Plans *****/

+!do_RTO : true

  <- .my_name(MyName);
  .print(MyName ," starting ...");

  !environment_ready;
  !observe_tags_board;
  !listen_for_optimizations
.

+!environment_ready : true

  <- !setup_tags_board;
  !setup_optimizations_stage;
  !setup_optimizers_ranking;
  !setup_control_panel
.

+!setup_tags_board : true

  <- ?process_tags(Tags);

  ?tags_datasource(

    url(Data_URL),
      user(Data_User),
      password(Data_Pass),
      dataset(Dataset)
  );

  ?ss_detection_problem(

    tags(SSDetection_Tags),
    time_unit(Time_Unit),
    sampling_start_delay(Sampling_Delay),
    sampling_freq(Sampling_Freq),
    time_windows_size(Time_Windows_Size),
    ss_detection_start_delay(SSDetect_Start_Delay),
    ss_detection_freq(SSDetect_Freq),
    ss_detection_method(SSDetect_Method),
    ss_detection_use_recent_date(SSDetect_Use_Recent_Date)
  );

  ?instant_profit_calculation(

```



```

        class(Profit_Calc_Class)
    );

    makeArtifact("TAGS_BOARD","br.usp.pqi.rto.art.TagsBoard", [],
        Board_Id);

    setupBoard(Data_URL, Dataset, Data_User, Data_Pass, Tags,
        Time_Unit, Sampling_Delay, Sampling_Freq,
        Time_Windows_Size, SSDetection_Tags,
        SSDetect_Start_Delay, SSDetect_Freq,
        SSDetect_Method, SSDetect_Use_Recent_Date,
        Profit_Calc_Class) [artifact_id(Board_Id)];

    startSampling;

    .print("Artifact " , "TAGS_BOARD" , " setted up: " , " (" ,
        Board_Id, ")");
.

+!setup_optimizations_stage : true

<-  makeArtifact("OPTIMIZATIONS_STAGE",
        "br.usp.pqi.rto.art.OptimizationsStage",[],Stage_Id);

    .print("Artifact " ,"OPTIMIZATIONS_STAGE"," setted up: " ," (" ,
        Stage_Id,")");
.

+!setup_optimizers_ranking : true

<-  makeArtifact("OPTIMIZERS_RANKING", "br.usp.pqi.rto.art.
    OptimizersRanking",[], Ranking_Id);

    ?performance_counters(Counters);

    setupRanking(Counters) [artifact_id(Ranking_Id)];
.

+!setup_control_panel : true

<-  makeArtifact("CONTROL_PANEL", "br.usp.pqi.rto.art.ControlPanel
    ",[], Control_Panel_Id);

    ?tags_datasource(url(Data_URL), user(Data_User), password(
        Data_Pass), dataset(Dataset));

    ?setpoints(SetPoints);

```

```

    ?feasibility_checkers(Checkers);

    setupControlPanel(Data_URL, Dataset, Data_User, Data_Pass,
        SetPoints, Checkers) [artifact_id(Control_Panel_Id)];

    focus(Control_Panel_Id);

    .print("Artifact " , "CONTROL_PANEL", " setted up and focused: ",
        " (" , Control_Panel_Id, ")");
.

+!observe_tags_board : true

    <- ?artifact_id("TAGS_BOARD", Artifact_Id);
    focus(Artifact_Id);
    .print("Observing artifact ", "TAGS_BOARD", " (" , Artifact_Id, ")
        ")
.

+!listen_for_optimizations : true

    <- ?artifact_id("OPTIMIZATIONS_STAGE", Artifact_Id);
    focus(Artifact_Id);
    .print("Observing artifact ", "OPTIMIZATIONS_STAGE", " (" ,
        Artifact_Id, ")")
.

+?artifact_id(Artifact_Name, Artifact_Id) : true

    <- lookupArtifact(Artifact_Name, Artifact_Id)
.

-?artifact_id(Artifact_Name, Artifact_Id) : true

    <- .wait(10);
        ?artifact_id(Artifact_Name, Artifact_Id)
.

+optimization_staged(Optimization) : poll_strategy(first_available)

    <- cartago.invoke_obj(Optimization, getSource, SourceAgentID);
    .print("Optimization staged by agent ", SourceAgentID);
        incrementCounter("staged_opts", SourceAgentID);

    checkOptimization(Optimization, Is_Feasible);
    .print("Optimization checked");

```

```

if (Is_Feasible) {

    .print("A feasible Optimization was found by agent ",
          SourceAgentID);
    cartago.invoke_obj(Optimization, getVarValues,
                      Var_Values);

    !update_setpoints(Var_Values);
}

else {

    .print("The Optimization found by agent ", SourceAgentID, "
          is not feasible to apply.");
    incrementCounter("dnf_opts", SourceAgentID);
}

.

+!update_setpoints(Var_Values) : process_state("steady")

<- .print("I do can update setpoints as process is steady");
updateSetPoints(Var_Values);

.

+!update_setpoints(Vars) : process_state("transient")

<- .print("Was going to update setpoints but process is transient");

.

```