Guilherme Simões Calado de Brito

Licenciado em Ciências da Engenharia
Electrotécnica e de Computadores

Service Broker module to support composition of loosely coupled components for process/product Servitization

Dissertação para obtenção do Grau de Mestre em Engenharia
Electrotécnica e de Computadores

Orientador: José Barata de Oliveira, Professor Doutor, FCT-UNL

Co-orientador: Giovanni di Orio, Mestre, UNINOVA/CTS

Júri:
Presidente: Prof. Doutor Fernando José Almeida Vieira do Coito
Arguente: Prof. Doutor João Almeida das Rosas
Vogal: Prof. Doutor José António Barata de Oliveira

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September 2017

Service Broker module to support composition of loosely coupled components for process/product Servitization

To my daughter and my family

Acknowledgments

Firstly, i'd like to express my gratitude to my supervisor, Professor José Barata, who gave me the possibility to develop this dissertation in my area of interest, and above all, the opportunity to initiate my research career being involved in FP7 ProSEco project, which leveraged my academic and personal growth.

To my co-supervisor, Giovanni di Orio, for all the orientation, teaching and support along these two years of development of this project. I'll be forever grateful for having him as a reference but most of all as a reliable friend, who is an honour to have as.

A very special thanks to André Rocha who, as my professor, not only lead me to this path, but mostly as my co-worker in the R&D group, where he always provided me a huge help with the most availability.

Also to my colleagues and friends from the R&D group, for all the help and good moments we passed: Pedro Monteiro, Mafalda Rocha and Ricardo Peres. For almost two years, everyday around you always brought motives to laugh while carrying on with our work.

To all my family, brother André, and specially to my parents. There are not enough words that can express everything they thought me, and all the support they've given. My mother for the daily love, patience and motivation to carry on this way. To my father, for being my role model in life.

Finally, a special thanks to my wife, Catarina, for all the support, dedication and tolerance given, which provided me the strength to move forward at the most difficult times, and the biggest thanks to my daughter Ândria, for everything.

Abstract

This dissertation presents the work done under the scope of the FP7 ProSEco project, regarding the design and development of the *Service Broker* module as foundation to support the runtime execution of composition of loosely coupled resources as services. ProSEco system stands as collaborative ICT solution, based on a Service Oriented Architecture, that enables companies the design and deployment of product-services (Meta-Products) as extension of their own products/processes, using Ambient Intelligence (AmI) technology, lean and eco-design techniques and applying Life Cycle assessment techniques. For this, ProSEco consists of two platforms, one dedicated to the design of the Meta-products, and the other that offers the deployment environment, based on distributed web-services that working together aim to fulfil the specifications provided on the design. The interoperability between these resources (services) is defined through usage of the *Service Composition* paradigm. As so, added development of generic features related to the *Service Composition* and to the interoperability of the involved services was made, enabling all the components and processes of the deployment to cope along the achievement of the objectives. Thus, the *Service Broker* module incorporates an agent-based system that is responsible for interpreting the specifications of the design products and of the Service Composition, and to manage the necessary resources (services) to be used on the deployment. From the side of the resources, the development intents to fulfil the *Service Composition* specifications, therefore, being adopted of generic aspects of development.

The resultant prototype was validated in virtual and real environments, supported by industrial application scenarios, and experimental results from one of them are also presented.

Keywords: Product Extension Services, Meta-Products, Service Composition, Service Oriented Architecture, Agent-based system

Resumo

Nesta dissertação é apresentado o trabalho realizado no âmbito do projecto FP7 ProSEco, relativo ao desenvolvimento do módulo *Service Broker* como fundação de suporte da execução de composição de recursos soltamente acoplados. O sistema ProSEco coloca-se como uma solução ICT colaborativa, baseada numa Arquitetura Orientada a Serviços, que possibilita às empresas industriais o desenvolvimento e execução de produtos-serviços (Meta-Produtos) como extensão dos seus próprios produtos/processos físicos, recorrendo a tecnologia Inteligência Ambiente, técnicas de *Lean* e eco-*design* e, aplicação de técnicas de avaliação de Ciclo de vida. O ProSEco consiste de duas plataformas, uma dedicada ao desenho de tais Meta-Produtos, e outra que oferece um ambiente de execução, baseado em serviços-*web* distribuídos que, em trabalho conjunto, apontam à concretização das especificações fornecidas no desenho. A interoperabilidade entre os recursos é definida através do uso do paradigma *Composição de Serviços*. Como tal, desenvolvimento adicional de características genéricas relacionadas com a *Composição de Serviços* e com a interoperabilidade dos serviços envolvidos foi feito, possibilitando que todos os componentes e processos da execução colaborem ao longo do cumprimento dos objetivos. Assim, o *Service Broker* incorpora um sistema de agentes que é responsável por interpretar as especificações do desenho dos produtos e da Composição de Serviços, e pela gestão dos recursos necessários. Do lado dos recursos, o desenvolvimento deve cumprir as especificações da Composição de Serviços, e como tal, são adotados com aspetos genéricos de desenvolvimento.

O protótipo resultante foi validade em ambientes virtuais e reais e suportado por cenários de aplicação, sendo que resultados experimentais de um deles são apresentados.

Palavras-chave: Serviços de Extensão de Produtos, Meta-Produtos, Composição de Serviços, Arquitetura Orientada a Serviços, Sistemas de agentes

Table of Contents

Table of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

ACL	Agent Communication Language
AmI	Ambient Intelligence
API	Application Programming Interfaces
B2B	Business to Business
BMS	Bionic Manufacturing System
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CPF	Cyber-Physical Features
DEOM	Design, Evaluation and Operation Methodologies
DMS	Dedicated Manufacturing System
EAS	Evolvable Assembly System
EPS	Evolvable Production System
FIPA	Foundation for Intelligent Physical Agents
FMS	Flexible Manufacturing System
GUI	Graphical User Interface
HAV	High-Adding Value
HMS	Holonic Manufacturing System
IaaS	Infrastructure as a Service
ICT	Information and Communication Technologies
IDE	Integrated Development Environment
IoT	Internet of Things
IT	Information Technology
JADE	Java Agent Development Framework

JAXB	Java Architecture for XML Binding
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JSON	JavaScript Object Notation
KMB	Knowledge and Management Base
LCA	Life Cycle Assessment
MAS	Multi Agent System
NIST	National Institute of Standard and Technology
OT	Operation Technology
P&P	Plug and Produce
PaaS	Platform as a Service
PES	Product Extension Service
PLM	Product Lifecycle Management
PSS	Product-Service System
QoS	Quality of Service
R&D	Research and Development
REST	REpresentational State Transfer
RMS	Reconfigurable Manufacturing System
SaaS	Software as a Service
SLM	Service Lifecycle management
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discover and Integration
WS	Web Service
WS-CDL	Web Service Choreography Definition Language
WSDL	Web Services Definition Language
XaaS	Everything as a Service
XML	eXtensible Markup Language

1

Introduction

1.1. Background

For the past few years, the industry modernization has forced manufacturing companies into committing in uplifting their value by evolving in new business models that go beyond the manufacturing and supply of physical products, steering into the addition of integrated solutions that enhance functionalities and services to these products. European manufacturing followed this strategic trend supported by the denominated *Servitization*, which was firstly introduced by (Vandermerwe and Rada 1988). In this context, Product-Service Systems (PSS), have boosted with great impact in the last years, as confirmed in (Qu et al. 2016), where is stated that PSS design, evaluation and operation methodologies (PSS-DEOM) have met an increased research on the time lapse of 2008-2015. PSS emerged as a way for manufacturing companies to impose more competitiveness against the strengths from competitors, such as the ones who act on low-wage regions, and also to keep in line with the new markets and business tendencies, which are forcing the manufacturing companies to place their efforts in innovation of products and services. High-

Adding Value (HAV) manufacturing steps forward over cost-oriented manufacturing, enhancing the ability to create innovative products (Meta Products) which are developed and delivered as SW services (Rubino, Hazenberg, and Huisman 2011), as result of ICT solutions developed for the purpose, and out coming as Product Extension Services (PES) around the physical products. The concept is enforced by the idea of, nowadays, most of manufactured products including Cyber-Physical Features (CPF), such as sensorial data and intelligent features, which tend to be explored with different, innovative and personalized services (Scholze, Correia, and Stokic 2016).

A Joint effort from manufacturing companies, technology companies and R&D Centers is being done on the development of such ICT Solutions that allows the design of effective PES, by means of the most advanced and reliable technologies and concepts available, and, allowing the integration and usage of new services adjustable to the CPFs in use on the current days.

The work presented in this dissertation regards to an ICT Solution (ProSEco Consortium 2014)oriented to provide the design and posterior deployment of PES Solutions based on a Service Oriented Architecture (SOA), that allows an effective product *Servitization*, as stated in (Di et al. 2016).

1.2. Research Problems

Considering the description given in Section 1.1, a challenge risen for the manufacturing world is creating ICT solutions capable of supplying a collaborative environment for development and deployment of new PES to the customers and for their own benefit, by using the most innovative technologies and techniques. ProSEco project tries to cover up this, by offering an ICT Solution which allows the integration of independent software solutions with specific objectives, and whose combination may provide innovative results to the industry. The ProSEco Solution is composed by two platforms dedicated to the Development of PES and to the posterior Deployment of PES, respectively. The first offers a collaborative environment that leverages the design of PESs oriented to the latest technologies, such as Ambient Intelligence (AmI) technology, Lean, Eco-design and Life Cycle Assessment techniques. On the other hand, the latest offers a service-oriented environment where the designed PES may be executed, that is, the platform hosts several different services which combined functionalities is able to generate unprecedented new services, in accordance to the designed PES.

Is visible that ProSeco concept already tries to address and solve some emergent research problems, but looking more in depth, the main questions brought up to surface are:

- How is it possible to specify the combination of distinct types of services/functionalities that may work/interact between each other towards achieving a specific result?
- Which architecture and available technologies are able to interpret and use these specifications of services/functionalities in order to execute them as planned, whilst preventing failures most efficiently?

For the first, a proposed hypothesis is to make use of the *Service Composition* paradigm, applying it to the system, and thus, implementing the resources/functionalities as services. The *Service Composition* enables the combination of the services by defining the interoperability terms between them, while not interfering with their specific implementation.

As for the second, a Service Oriented Architecture is considered, where services specific implementation and the associated interoperability may be executed, in a real-time environment. For doing so, the architecture encompasses several components, that are able to receive the designed specifications regarding the resources specific implementation and the *Service Composition*, and further execute it according to the plan. An internal agent-based system is proposed to perform the interpretation of the *Service Composition* specifications, and engaging the necessary resources dedicated to each PES.

1.3. Hypothesis overview

Having identified the research problems, a hypothetic solution is proposed. Since the whole concept encompasses several concepts and features, the Hypothesis proposes to separate the problem in minor parts, and focus on a solution for each of the distinct problem or component, and further joining efforts into a complete final solution.

For the first research problem presented on Section 1.2, the implementation of an Engineering Tool dedicated to the composition of services, as part of the PES development process was considered. This Engineering Tool (*Service Composition Engineering Tool*) shall offer a graphical environment where the users may combine the resources selected for a given PES and, furthermore, provide the parameterization of the interoperability that is desired for the PES deployment. This implies the creation of a Meta-language that reproduces the given specifications, and that is understood by other components of the system, especially the ones that are dedicated to interpreting and use such features, as the *Service Broker* or the resources that execute the PES.

As for the second problem, the considered architecture of the environment dedicated to the PES Execution needs to encompass the necessary modules, based on a Service Oriented Architecture

(SOA). The fundamental component of SOA based platform is the *Service Broker*, which must be capable of discovering the available services and/or mediate the connectivity between the services. Thus, the Services that offer the functionalities (in this case the resources to be used for executing the PES and provide results) must be reachable by the *Service Broker*. For the considered architecture, a *Service Registry* provides an accessible database where this information can be obtained. Also, the platform is able to launch and host the resources through the implemented module *Deployer*, which may be distributed on different machines of the network, but still discovered and invoked by means of the *Service Broker* action. In relation to the *Service Composition*, the *Service Broker* must be able to receive the PES (software structure data encompassing all the resources specifications, as well as the *Service Composition* specifications provided during the PES development), and make all the arrangements for starting the PES execution. For this, an agent-based internal system was considered with the purpose of, according to *Service Composition*, allocate, setup, command and monitor the demanded services that will work on the PES Execution.

Finally, the own Services need to adopt generic features and methodologies that abstract them from their specific implementation and, also, make them aware of the *Service Composition* specifications that regard to the designed interoperability between them. This also allows that new possible types of resources to be developed in the future, may be integrated in the system and cooperate in the execution of new PESs.

Satisfying and combining all these individual conditions, is possible to achieve the main objective of having a platform provided of autonomous components and automatized mechanisms dedicated to the full process of deployment of PES, and capable of following the specifications designed by the users. The resultant prototype was tested and validate, assured by both virtual and real industrial application scenarios, by means of developing and further executing PES with specific objectives and using real data extracted from machines.

1.4. **Dissertation Outline**

This dissertation is organized as follow:

- Chapter 2: presents the state-of-the-art of the application of Product-Service Systems on manufacturing together with supporting concepts that have driven the design and development of ProSEco prototype
- Chapter 3: presents the overall concept and general architecture of ProSEco system, and further detailed description of the *Service Composition* features, *Service Broker*

implementation and adopted applied generic features and methodologies of the resources that boosted the full functionality of the system

- Chapter 4: describes all the relevant works done in implementation, based on the proposed architecture
- Chapter 5: assessment and validation by presenting a real application scenario
- Chapter 6: summarizes the main conclusions and contributions of the work presented in this dissertation and possible improvements to be addressed in the future.

2

State-of-the-Art Analysis

2.1. Manufacturing Paradigms

Since the beginning, manufacturing industry has always been conditioned by social, economic and technological aspects that influenced the organization of the manufacturing processes. In response to the involving market demands and society conditioning changes, the industry has been forced to evolve, by developing new manufacturing processes to produce products which triggered the raise and death of several business paradigms that, according to (Yoram Koren 2010), intended to provide new ways to sell them. Business paradigms evolved spread over three different ages: industrial age, Information age and Post-Information Age, as seen in Figure 2.1.

The start and end of both Business Paradigms and Ages are unclear, being impossible to specify exact points in the timeline, however it can be identified overlaps due to the progressive raise and abandonment of the adopted Paradigms. A very detailed and comprehensive overview on manufacturing paradigms, with a complete analysis over their evolution and related aspects can be found in (D. A. C. da G. Barata 2015; Luis Ribeiro and Barata 2011; Yoram Koren 2010).

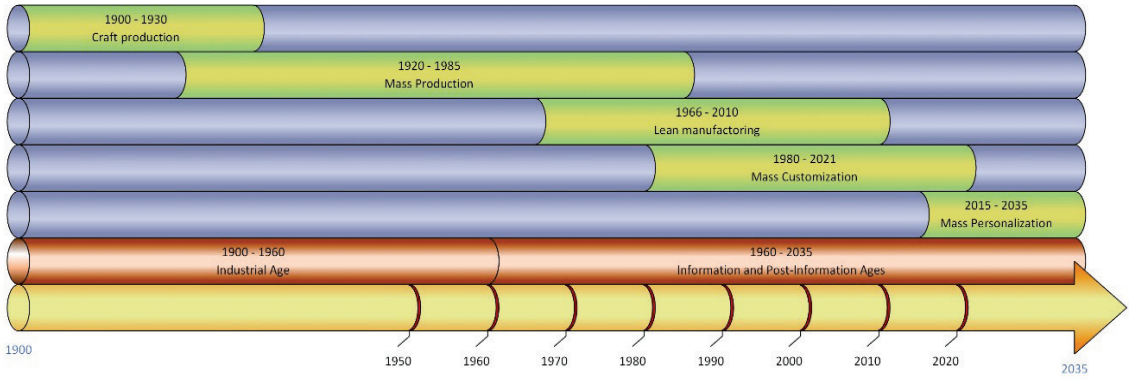

Figure 2.1 - Manufacturing business paradigms until the present day (Di Orio 2013; Oliveira 2003)

2.1.1. Mass Customization

Since the 1980s, economy growth and society rising wealthier had an impact on a higher demand for larger variety of products to choose from, as the standard low-cost products are not enough to satisfy the customer's needs. However, more recently, the continued increase in the demand for customized products, often to the extreme (a different product for each customer), which are in turn getting more and more complex and varied in regards to their application domain, has translated into shorter changeover times and product life cycles, moving further and further away from the idea of standardized mass production, towards mass customization instead (Nagorny, Colombo, and Schmidtmann 2012).

Mass customization paradigm was popularised by Joseph Pine II (Pine 1993), who describes it as the new way of doing business with its core set on quickly increasing variety and customization of products, without incrementing costs. The foreground challenges and requirements of this paradigm rests on identifying and fulfil customer's individual demands and desires without sacrificing efficiency and effectiveness while supporting low-cost, as opposed to previous manufacturing paradigms. This is transposed on the supply of standard products in mass, but with the capability of adding extra features and/or packages (usually as the ending assembly process), resulting on assorted products. A good example is the automotive sector, where the association of personalized packages over the same model is already a trend (Juehling et al. 2010).

Throughout the years, several manufacturing processes, based on the most diverse technologies, architectures, approaches and methodologies, have been designed and implemented to satisfy mass customization requirements, while improving and ensuring manufacturing companies competitiveness and positioning in market sharing. New business forms, reliant on a desire for strong collaboration between suppliers and customers, has imposed further challenges to the shop floor, making older approaches unsuitable for this new reality (Frei, Barata, and Onori 2007), and emerged new trends associated to the paradigm. Some of the most popular key words found in

literature for defining these trends are: collaborative, flexible, reconfigurable, lean, holonic, agile, evolvable, collaborative, adaptive or self-learning.

Flexible manufacturing Systems (FMS) emerged due to the manufacturer's need for accompanying the increasing need for product customisation and variety, therefore focusing on the production of diverse types of products, with changeable volume and mix, by adjusting the system through a priori built-in features controlled by reprogrammable equipment (Cândido 2013). However, it is considered that flexibility of FMS is lost when dealing with new and unexpected requirements, since the need for the system to automatically make the required adjustments is complex (Leitão 2004), which led to gradually being surpassed by other paradigms: Reconfigurable Manufacturing Systems (RMS) (Yoram Koren et al. 1999; Mehrabi, Ulsoy, and Koren 2000), Bionic Manufacturing Systems (BMS) (Ueda 1992; Okino 1993), Holonic Manufacturing Systems (HMS) (J. H. Christensen 1994; Van Brussel et al. 1998; Leitão 2004), and more recently Evolvable Production Systems (EPS) (Onori 2002; J. Barata, Santana, and Onori 2006; Neves and Barata 2009).

RMSs provide customized flexibility through scalability and reconfiguration as demanded by the market requirements in a more agile manner (ElMaraghy 2005), as a reaction to FMS shortages. The main principles of RMS include modularity, integration, flexibility, scalability and adaptability, that leverages a faster adaption to new products due to easiness on switching modules and fast reconfiguration depending on the production requirements, while integrating different technologies and variances in the demand, i.e., a RMS is designed to provide agility to proceed to changes in production capacity without affecting the overall robustness and reliability.

The BMS paradigm is inspired in living organisms, with a greater focus in natural organs, sharing some of the same base concepts, namely complexity encapsulation, self-organizing behaviours and decentralisation. The basic concept is centred in a hierarchy where the whole organism is composed by different interacting organs, dynamically exchanging data similar to DNA, turning the system as a whole to enact a self-organising response. When applied to manufacturing, this paradigm aims to provide solutions to deal with unpredictable changes in the production environment based on bio-inspired behaviours such as self-adaptation, self-organisation and the capacity to evolve as previously stated.

HMS is inspired on the work of Arthur Koestler (Koestler 1968), who proposed the word *holon* to describe the basic unit of organization in biological and social systems. HMS paradigm transposes the concepts that Koestler developed for social organization and living organisms into the manufacturing production system world, as stated by (Van Brussel et al. 1998). Each *holon* can be a part and a whole at the same time while being part of group referred as *holarchy*. Each

holon pursue a particular task or goal through coordination, cooperation and negotiation within a *holarchy*. When applied to a manufacturing context, the holonic concept results in a distributed system comprising several subsystems, each exhibiting holonic behaviour offering higher degrees of adaptability and scalability. The goal of HMS is to take advantage of the inherent benefits: stability while facing disturbances, adaptability, flexibility and responsiveness. An extensive review about HMS can be found in (Babiceanu and Chen 2006).

The Evolvable Assembly/Production Systems (EAS/EPS), uses the aggregation of many small modules that provide simple functionalities, adopting the system with a dynamic self-adaptation to new products, processes and production scenarios, while enabling the evolution of the system together with the surrounding environment, such as adding or removing manufacturing modules in response to changes in production orders and plans at run-time without the need to stop the system for reprogramming or reconfigure the process tasks. Thereby, EAS/EPS proposes a solution which, being based on many simple, re-configurable, task-specific elements (systems modules), enables for a continuous evolution of the assembly/production system (Onori, Barata, and Frei 2006). Modules are abstracted as process specific entities rather than function specific, as it happens in RMS for instance (Luis Ribeiro and Barata 2011), however EPSs can achieve fine granularity if needed.

2.1.2. Mass Personalization: New Manufacturing Trend

With the appearance and global dissemination of the World Wide Web, the doors were opened to the Post-Information age, by changing the way people and manufacturing companies interact. The information has become more and more personalized, while accessibility became easier everywhere and to everyone.

2.1.2.1. Emergence scenario

In the last years, customer's demand for more customized and personalized products is leading into a change on companies' business and operations strategies, as customers' are tending to, each more and more, be provided of: high quality, fast delivery and, most of all, a higher level of product customization, yet unwilling to pay the commensurate price (Kumar 2007). In response to this, the adopted strategies are steering into the realization of profit from a *market of one*, that is, production of products is getting each time more conducted by the customers' demands, needs, and thus, confirming the tendency for increasing the variety of product. This novel approach that aims to provide the most of individual personalization of products is steering the manufacturing paradigm from mass customization into mass personalization.

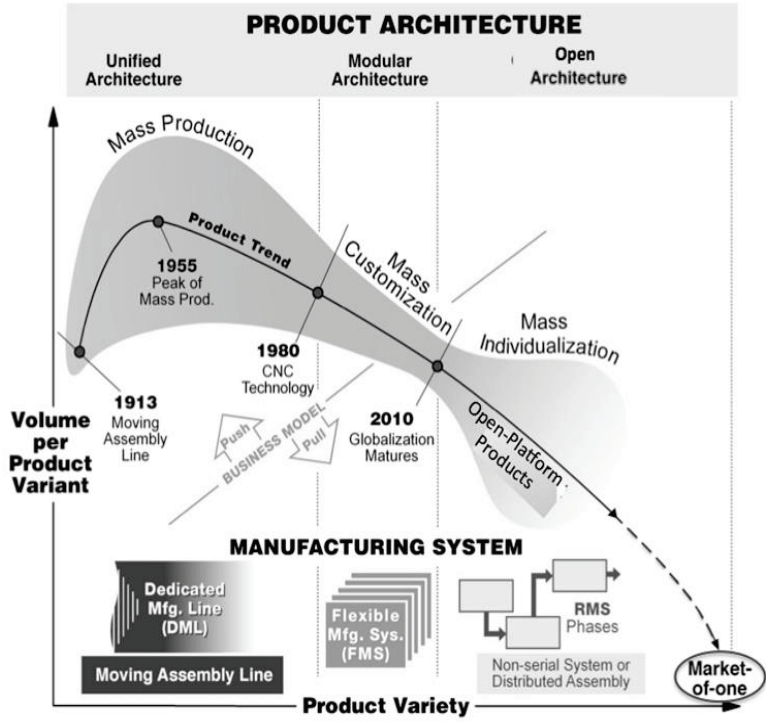

Figure 2.2 - Transition of manufacturing paradigms in the last 100 years; volume of each product variant is going down, and with open-products may reach a Market-of-One (Y. Koren et al. 2015)

The degree of transformation of a company is related to the extent which its product is soft, i.e., can be electronically produced. Thus, at the bottom of the personalization spectrum are manufacturing companies engaged in producing hard, configurable products, while on the high end of the spectrum are service companies whose product can be totally configured and delivered electronically (Kumar 2007). Manufacturing processes for design should then be as flexible and agile as possible, therefore a decoupled production process is needed in order to handle high volumes of product with a wide variety, for satisfying all kinds of customers and face the markets turbulence and unpredictability (Jassbi et al. 2014). From the customer point of view, the *mass personalization*, also referred as *mass-individualization* by in (Y. Koren et al. 2015), he/she shall be provided of a more involving, active and even interactive role on the design process of products, therefore, manufacturer companies must evolve in order to provide the necessary means for making it possible.

Differentiating mass customization from mass personalization, the first is based on modular product architecture, where the modules are designed by the product manufacturer, creating a (possibly very) large of optional product choices, and where the customer may choose which ones to include on the final product, which is finally assembled/produced and delivered. While on the latest, the manufacturer provides the product platform, adopted of several interfaces that enable the integration of new modules, allowing the customer to search for the desired one. This integration approach allows that new modules can be produced and inserted in the platform by

other vendors than the manufacturer, therefore contributing for the possible growth of other companies. The customer gets to be involved on the design process of his personal product composed by the platform and the selected modules, which only after is sent to the manufacturer and produced.

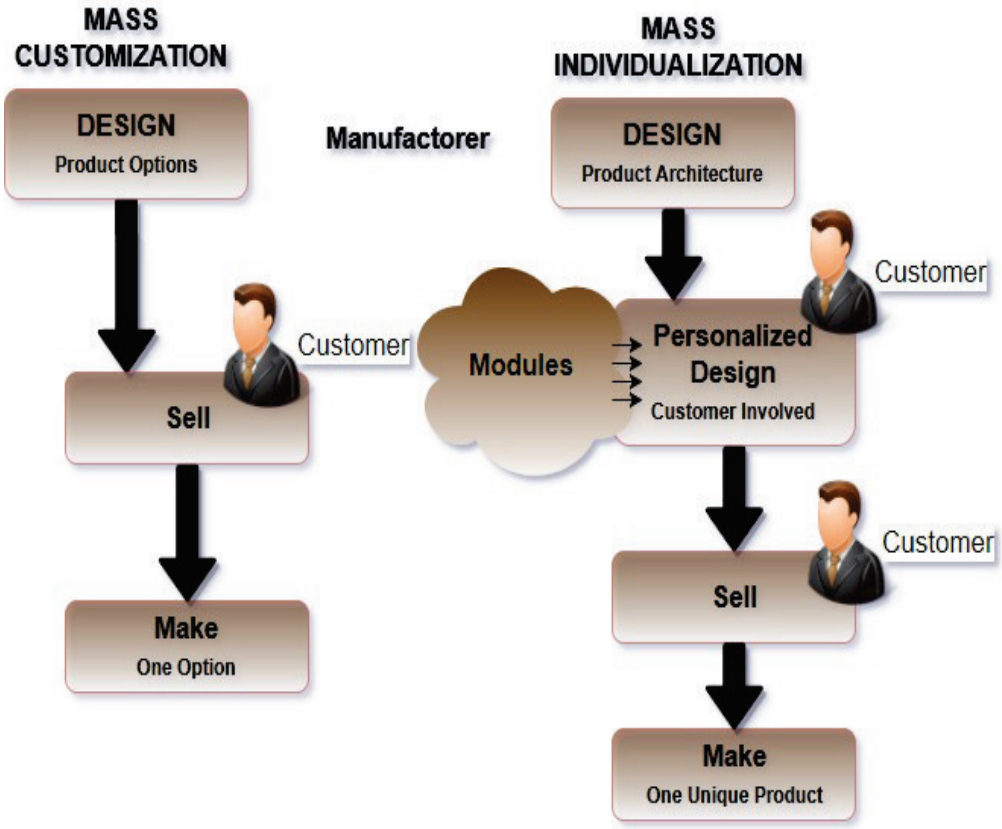

Figure 2.3 - Flow-chart of mass-customization and Open Product (Y. Koren et al. 2015)

To capitalize on the key markets opportunities and winning the competition for markets share, manufacturing companies are caught between the growing needs for:

a) implementing more and more exclusive, efficient and sustainable production systems to assure a more efficient and effective management of the resources and to produce innovative and appellative customized products as quickly as possible with reduced costs while preserving product quality;

b) and creating new sources of value by providing new integrated product-service solutions to the customer (Cavalieri and Pezzotta 2012).

Aiming at the fulfilment of these demands, it's being progressively deducted that manufacturing companies need to be internally and externally agile by among the several structural areas, from devices data management settled on the shop floor, and rising to business data management are even extending to beyond the individual company frontiers, to intra-enterprises data management at organization level. Therefore, agility implies being more than simply flexible and lean (Cândido

2013). Flexibility refers to the ability exhibited by a company that is able to adjust itself to produce a predetermined range of solutions or products (Sethi and Sethi 1990), while lean essentially means producing without waste (Shah 2003). On the other hand, agility relates to operating efficiently in a competitive environment dominated by change and uncertainty (Goldman 1995).

An agile manufacturing company shall be capable of detecting rapidly fast changing needs of the marketplace and propagate these needs to the lower levels of the company in order to shift quickly among products and models or between products (Yusuf, Sarhadi, and Gunasekaran 1999), by applying a top down enterprise wide effort that supports time-to-market attributes of competitiveness. Thus, to be agile, a manufacturing company needs a totally integrated approach, combining integrate product and process design, engineering and manufacturing with marketing and sale in a holistic and global perspective, which is not yet properly covered in manufacturing companies of today.

As stated in (Colombo et al. 2014), several efforts have been made towards structural and architectural definition and characterization of a manufacturing company and its production management system. Among others, the most popular and still practical applied is the set of definitions embodied into the ISA-95/IEC62264 standard:

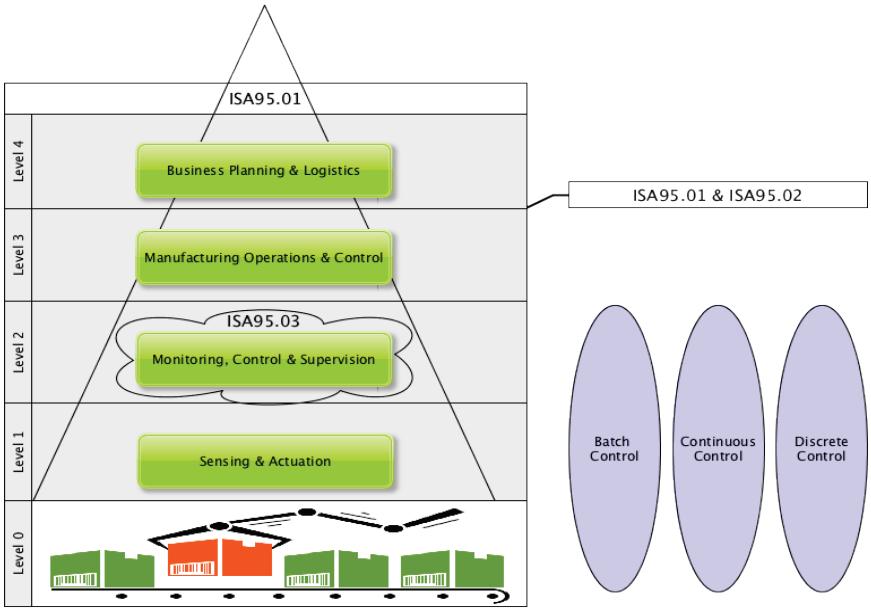

Figure 2.4 - Manufacturing company functional hierarchical decomposition according to the ISA-95/IEC62264 standard

According to this standard, standard manufacturing companies and their production systems (process plus factory) are organized into a five-level hierarchical model also known as "automation pyramid". The standard also provides a set of directives and guidelines for

manufacturing operations management such as primary & secondary processes, quality assurance, etc., but even it being the widest used approach for modelling manufacturing companies, nowadays it does not show all the intricacies of the applications, the communication protocols, and – more in general – of the several solutions present at each one of the five levels. As a matter of fact, heterogeneity in terms of hardware and software – as well as – data distribution (transmission of information from several signal sources) and information processing are not fully covered by the ISA-95/IEC62264 standard. Even though it defines an information exchange framework to facilitate integration of business applications with the manufacturing control applications within a manufacturing company (Dassisti et al. 2008), the lower levels of the pyramid are not addressed, implying that the automation pyramid – as it is – has significant limitations regarding the increased complexity of modern networked automation systems (Pratl et al. 2007), in particular when it's used to support:

a) the integration of modern technologies and devices, and their lifecycle management;
b) the handling of the information flow along the overall automation pyramid from the lower level to the higher ones (company visibility);
c) and the handling of the information flow coming from intelligent devices spread all over the living environment that could be used as fundamental feedback shared inside the automation pyramid.

Concrete instantiations of the ISA-95/IEC62264 standard confirm the above issues, as state-of-the-art industrial automation solutions are known for their plethora of heterogeneous equipment encompassing distinct functions, form factors, network interfaces and I/O specifications supported by dissimilar software and hardware platforms making the process of integrating new technologies and devices extremely complex and expensive while promoting the design and development of new tools and services to assist and reduce the effort, cost and delay of lifecycle (re)engineering interventions. Moreover, the information and communication technology (ICT) is a core element and fundamental infrastructure for manufacturing since it supports all the technical developments, management systems, administration, business and manufacturing processes. However, as explained in (Orio et al. 2015), manufacturing companies are not fully benefitting from their ICT infrastructure due to the heterogeneity of their state-of-the-art automation solutions that in turn leads to the often-poor cross-layer integration, since poor communication between the distinct layers of the ISA-95/IEC62264 standard is provoked by the lack of integration between operation technology (OT) and information technology (IT) in a company. This disconnection between OT and IT backbone infrastructure implies that the data generated by the field equipment is rarely used within business and vice-versa. As stated in (Davidsen 2014), the business leaders of today are becoming more and more conscious about the possibilities and opportunities of gaining access to both production and product data, which can

be used for identifying important production events, predicting market fluctuations, savings costs, improving the overall quality of the products, optimizing the production processes, optimizing product design phase while enabling high product customization and *servitization*. Furthermore, the disconnection between OT and IT is not the only obstacle to an effective and efficient usage of data, as nowadays the product itself is a fundamental data source that is not adequately exploited. Therefore, data – generated by processes and products during their operation – allows manufacturing companies to transform information into insight and more generally into knowledge to be used for the sake of responsiveness, (re-)configurability, adaptability, and transparency of processes/products and global reach of the business (Davidsen 2014), that is to say, for improved agility that covers all the layers of the ISA-95/IEC62264 standard – from management to shop-floor – while encompassing the overall product/service lifecycle management (P/SLM).

Thus, an agile manufacturing enterprise should be capable to detect the rapidly changing needs of the marketplace and propagate these needs to the lower levels of the enterprise in order to shift quickly among products and models or between products, by using a totally integrated approach for product and process design, engineering and manufacturing with marketing and sale in a holistic perspective. Putting the light on the "data", there is a tremendous need for reference model and architecture to be used as the basis for virtualization and decentralization of the overall ISA-95/IEC62264 pyramid while enabling the integration of data coming from "connected" products, and so, the integration and use of data from all levels must be facilitated.

Current technological trends in both industrial and living environments are pushing more and more to the idea of pervasive and ubiquitous computing while offering – at the same time – a huge opportunity to link information sources to information receivers/users. Future internet technologies – such as Internet of Things (IoT) and Cyber-Physical Systems (CPS) – facilitate the deployment of advanced solutions in plant floor, as well as, day to day applications while promoting the meshing of virtual and physical devices and the interconnection of products, people, processes and infrastructures within the manufacturing value chain. The deployment of IoT/CPS-based systems (such as MANTIS[1] or ProaSense[2] projects) is enabling the creation of a common virtualized space to facilitate the data acquisition process across multiple heterogeneous and geographically distributed data sources, and further use of this data. It is necessary to comprehend that today's problem is no longer networking (protocols, connectivity, etc.) nor it is hardware (CPU/memory power is already there, at low-cost and low-power consumption) but

[1] http://www.mantis-project.eu/
[2] http://www.proasense.eu/

rather it is on how to link disparate heterogeneous data sources to the specific needs and interaction forms of applications and platforms. Under such circumstances, the main challenge is the interoperability between information sources and receivers/users. Being able to combine all these technologies is a challenge whose resolution has the open door through a recent Information and Technology (IT) paradigm that stands as the propulsion of mass personalization - Cloud Manufacturing.

2.1.2.2. Cloud manufacturing: trigger for mass personalization

Cloud Manufacturing emerges has an extension of Cloud Computing (further detailed in section 2.2.1) applied to the manufacturing industry, in the sense of completing the demand for globalization by transforming manufacturing business in the direction of a new paradigm where resources and capabilities are componentized, integrated and/or optimized through a worldwide accessibility (Vincent Wang and Xu 2013). Thereby, the concern on the design process of products is split between the manufacturing company, who provides the architecture and basic modules and interfaces, and the customer who intervenes directly in the creation of his personalized product by selecting and composing the available services/features.

As stated in (Tao et al. 2011), in Cloud Computing, the resources are primarily computational resources (servers, databases, network, software, etc.), provided as services belonging to one the three different categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (Saas).

In Cloud Manufacturing, it differentiates from Cloud Computing as the resources of the system are manufacturing resources, that is, physical manufacturing devices, machines or sub-parts of the machines, that are abstracted in terms of their functionalities and capabilities which are provided to the user as services included in IaaS, PaaS or SaaS. A layered framework for implementing Cloud Manufacturing consisting of four layers can be considered (Xu 2012), as presented in Figure 2.5.

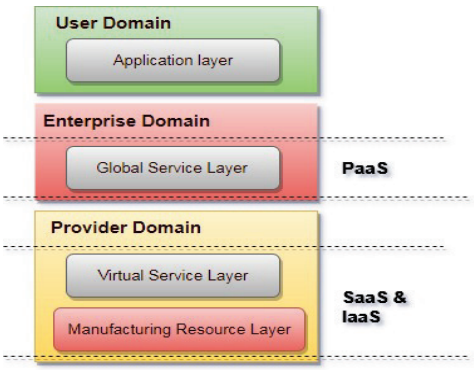

Figure 2.5 - Layered framework for implementing Cloud Manufacturing (Xu 2012)

Physical manufacturing resources and the shop floor capabilities that are provided to the user as SaaS and/or IaaS are included in the Manufacturing Resource layer. The virtual service layer is responsible for virtualizing the manufacturing resources and encapsulate them into cloud manufacturing services that in turn are provided to the Global Service Layer. Therefore, the Global Service Layer is responsible to manage the Cloud Manufacturing services. Finally, the Application Layer is the entry point of the manufacturing companies and provides to the user the possibility to build/construct manufacturing applications from the virtualized resources. Therefore, users are provided of services that encompass the whole life cycle of manufacturing: Design as a Service, Manufacturing as a Service, Experimentation as a Service, Simulation as a Service, Management as a Service, Maintenance as a Service, and Integration as a Service (see Figure 2.6).

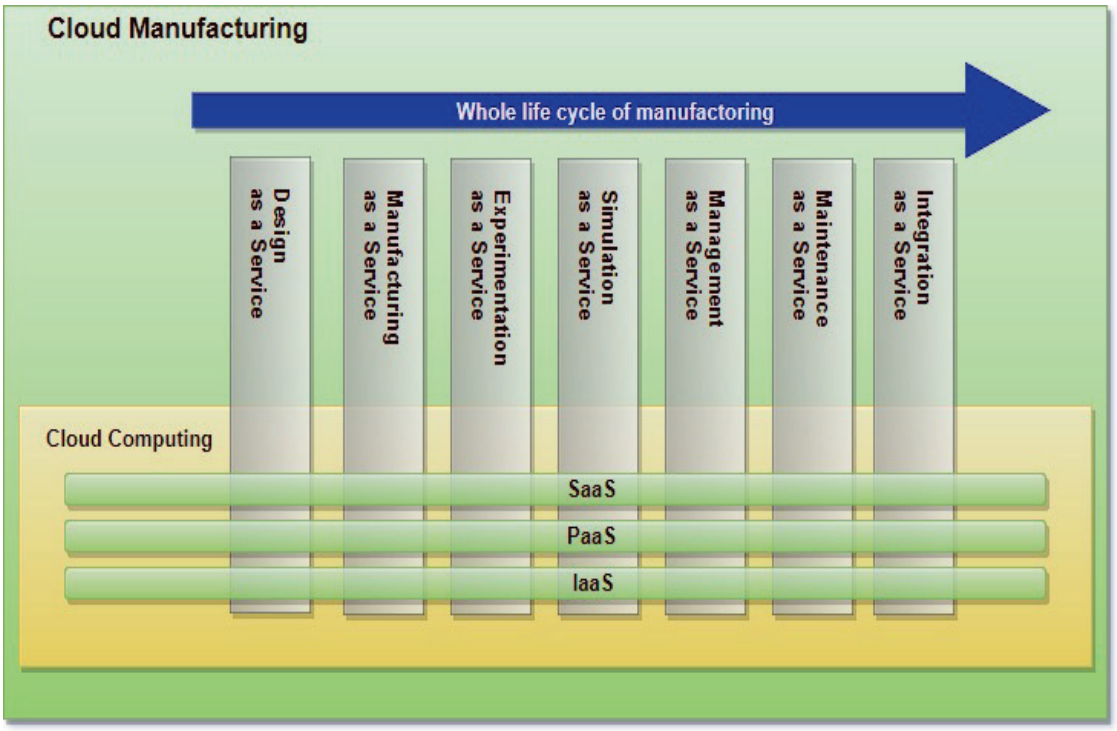

Figure 2.6 - Cloud Computing and Cloud Manufacturing in a nutshell

From cloud computing, several characteristics contribute for shaping the cloud manufacturing concept (Ren et al. 2017):

- **Service-centric perspective**: all resources are delivered as standardized services over the internet, enabling the possibility for outsourcing IT business to third parties or service providers in the cloud
- **Virtualization:** enables the decouple the tight binding between the Upper IT systems and underlying hardware infrastructure

- **Scalability and elasticity:** allowing computer resources to scale up and down according to the runtime workload, and provide the necessary quantity of resources, even in case of peak loads.
- **On-demand customization:** provisioning of methodologies for self-customization, discovery, configuration and deployment of new services

Also, as already stated, Cloud Computing is also motivated by the evolution of several other concepts and technologies proposed in the past, and at the current days stands as the de-facto paradigm that affords the means to accomplish the current demands, by leveraging the collaborative environments and integration of the most advanced technologies, in the form of services, that enables the design, simulation and production of custom made products designed by the customers, thus driving us towards the elevation of the mass personalization paradigm.

A more detailed and up-to-date characterization and examples of key applications can be seen on (Ren et al. 2015) and (Ren et al. 2017).

2.2. Supporting Concepts

2.2.1. Cloud Computing

Cloud computing merges as the last computing paradigm that promises flexible IT architectures, configurable software services, and QoS (Quality of Service) guaranteed service environments. Although the name was coined in 2007, the initial concept has been elaborated already in the 1960s, at the time relating with the delivery of computer resources over a global network (Licklider 1963). A more formal definition of this concept and/or paradigm was given by the National Institute of Standard and Technology (NIST), where Cloud Computing was defined as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." (Mell and Grance 2009). The concept translates as everything being abstracted as a Service (XaaS), from the platform application modules to the components of the software system. of the hardware, into the hardware itself. In Cloud Computing, services are structured in three layers of abstraction, according to the level of capability provided and the service model of the providers, namely: Infrastructure as a Service, Platform as a Service and Software as a Service (SaaS) (Chou 2010), as seen in the service delivery model of a typical cloud based system depicted in Figure 2.7.

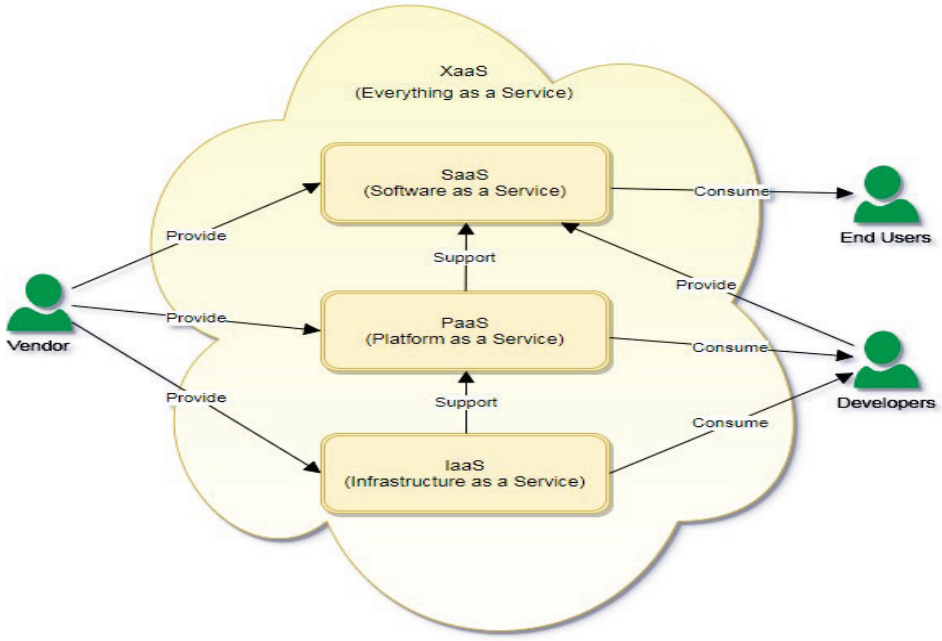

Figure 2.7 - Cloud Computing Service Delivery Model (Marinos and Briscoe 2009)

According to (Xu 2012), the IaaS supplies standardized services over the network defining processing, storage, networks and other fundamental computing resources. Cloud providers' clients can deploy and run operating systems and software for their underlying infrastructures. The middle layer, i.e. PaaS provides abstractions and services for developing, testing, deploying, hosting, and maintaining applications in the integrated development environment. The application layer provides a complete application set of SaaS. These services/applications can be accessed by users through Web portals, meaning that the end consumers are now using online services (Cloud) that can provide the same functionalities as a local computer application or program.

2.2.2. Product Service Systems

For some years, customers started to demand more than just a specific product, and instead, begun to ask for solutions for given problems. Companies continuously strive to increase production (Beuren, Gomes Ferreira, and Cauchick Miguel 2013), but then, while absorbing this new need, companies realized that providing physical products alone is not sufficient in terms of competitivity (Yu, Zhang, and Meier 2008), therefore new offers that aim at increasing market share and also customer's satisfaction have emerged (Sakao, Ölundh Sandström, and Matzen 2009; Sundin 2009). PSS has been used to accomplish this, by targeting companies' competitiveness and profitability increase, and by reducing the consumption of products through alternative scenarios of product use instead of purchasing. In reality, PSS concept has the potential

to re-orient the current standards of consumption and production, thus enabling a move towards a more sustainable society (Manzini, Vezzoli, and Clark 2001).

As stated by (Beuren, Gomes Ferreira, and Cauchick Miguel 2013), a commonly accepted definition of PSS is given by (Goedkoop et al. 1999), defined as a combination of products and services in a system that provides functionality for consumers and reduces environmental impact. The key elements of a PSS are: (1) the product; (2) the service, where some activity is done without the need for a tangible good or without the need of the system; and (3) the combination of products, services and their relationships.

Thereby, PSS sets a business model which brings benefits to companies, such as continuous improvements of the business, innovation in quality and satisfaction of consumer demand, by offering services that are extensions (Product Extension Services) around the products they provide. Another advantage is that information inherent to the products, and which customers are willing to supply back to providers, can be (re-)used to develop new systems and improve the product performance, and possibly improve the company's position in the value chain.

More recently, and focusing in the manufacturing industry, the new generation of products tend to include Cyber-Physical features (CPF), such as sensory systems and various intelligent features (Baines et al. 2007). These features, allied to the collected data from machinery, enable the idealization and design of Product Extension Services (PES), also called Meta-Products, with different innovative, personalized and context sensitive products and customer support services, such as is presented in (Scholze, Correia, and Stokic 2016), enabling the establishment/further enhancing of PSS.

The PSS approach can be very facilitating by ICT solutions (SW service), and can easily be integrated on the Cloud Manufacturing environment, seen on section 2.1.2.2.

2.2.3. Service Oriented Architecture

Service Oriented Architecture (SOA) (Erl 2005; Josuttis 2007; Papazoglou and van den Heuvel 2007), is a term of an emergent approach that addresses the requirements for loosely coupled, standard-based, and protocol-independent distributed computing, through promising architectural designs for rapid integration of data an business processes.

SOAs are viewed as one of the next evolutionary step to assure organization that meets more complex challenges imposed by globalization and market fragmentation, establishing an architectural model that targets to enhance efficiency, agility and productivity of an enterprise by positioning services as the building blocks, supported by a framework/platform for accomplishing

rapid system development and easy system modifications, while enhancing systems integration capabilities and overall system quality. In (Komoda 2006), SOA is defined as a design framework for construction of systems by combination of services and using profound ICT infrastructures as communication backbone. SOA is guided for the development and implementation of a platform that hosts independent services, which by their turn, represent well-defined and self-contained modules that provide specific operations, which are internally and/or externally accessible through standard interfaces, thus, enabled to be invoked by the consumers of the services. Furthermore, services can be (re-)combined into different application scenarios, since they are not dependable from state and/or context of the other services (Di Orio 2013). Therefore, the advent of SOA paradigm brings a radical change to the system components interaction leveraging *interoperability*, in the sense that standard based interfaces are applied to the different services, regardless of the implementing technology of each of them, and *scalability*, as services can be added or removed without interfering on infrastructure.

The existence of *Web Service*, has enabled and stimulated the implementation and development of SOA. However, as stated by (Natis 2003), *Web Services* do not necessarily translate to SOA, and not all SOAs are based on *Web Services*, but still there are irrefutable evidence that they are mutually and highly influenceable. Therefore, it's important to notice that *Web services* and *Services* are not the same. As stated in (Barry 2003), the term *Web Service* refers to a collection of technologies such as eXtensible Markup Language (XML) (Bray et al. 1997), Simple Object Access Protocol (SOAP) (Box et al. 2000), Web Services Definition Language WSDL (E. Christensen et al. 2001) and Universal Description, Discover and Integration (UDDI) (Bellwood et al. 2002). Therefore, *Web Services* provide a standard way of interoperability between different software applications, running on a variety of platforms and/or frameworks without being dependent of a particular hardware and/or technology, whereas a *Service* is what is connected using *Web Services*, meaning that they represent the endpoint of a connection (see Figure 2.8).

The methodology for providing and consuming a *Service* by means of *Web Services* technology is performed in the following sequence:

1. A Service Provider describes its service(s) using WSDL, and publishes these definitions on a Discovery Agency that supplies the ability for discovery of the registered services anywhere on the network;
2. A given client may query the Discovery Agency in order to locate the more suitable service that meets the query purposes;
3. The client receives the WSDL of the service provider(s) with the relevant information of where and how to connect to the given service;
4. The consumer sends the request to the service in accordance to the WSDL information

5. The Service provider (when invoked), executes the operation of the respective service and, if a request-reply is defined, it provides the result, also in accordance with the WSDL.

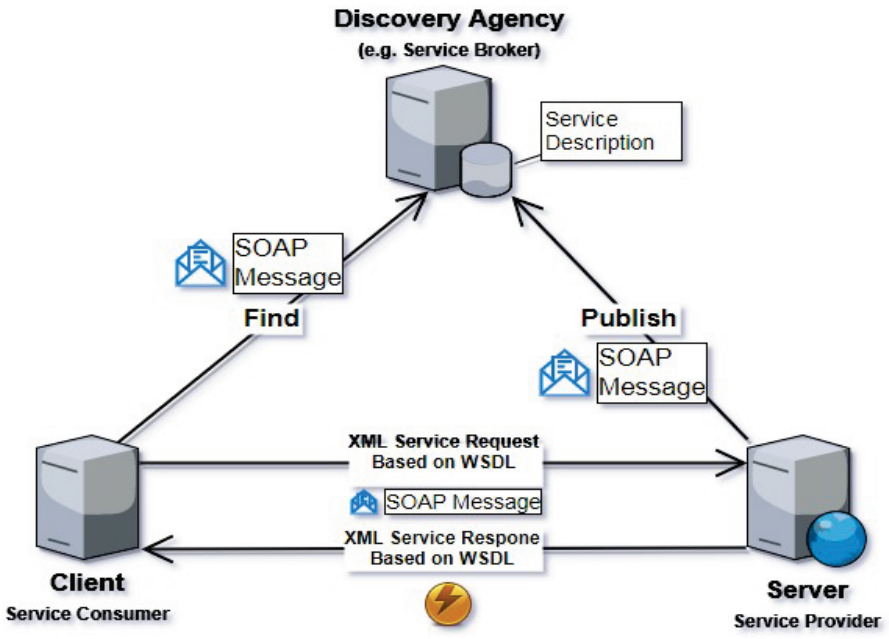

Figure 2.8 - Web Services basic components

2.2.4. Service Composition

As seen in the previous section, *Services* provides a mean for interactions between a client and a server provider, and constitute the building blocks of a SOA. However, in the urge to generate more complex functionalities, a new approach has arisen where the ability to combine and/or assemble atomic services is considered, in order to obtain the service abstraction mentioned in (Cândido 2013). In this scenario, the term *Service Composition* is defined in (Dustdar and Schreiner 2005) as the process of developing a composite service, which is obtained by the composition of the functionalities of several simplest services towards a defined objective.

According to (Peltz 2003), two main approaches are used for creating business models using the composition of Web Services: *choreography* and *orchestration*.

2.2.4.1. Service Orchestration

In the *Orchestration* approach, a central entity is responsible for controlling the workflow of the selected services, by completing the predetermined aspects of interaction between them, during the execution of the complex process. The workflow logic consists in a set of defined rules, conditions and events of the integrated system, that is, it specifies how the different services must interoperate with the central entity in order for it to perform the control and actions over them (see Figure 2.9). Therefore the Central Entity is the coordinator (as a Maestro of an orchestra) of

the entire composition while the services act as components, agnostic to the fact that they are part of a larger process (Bucchiarone, Melgratti, and Severoni 2007).

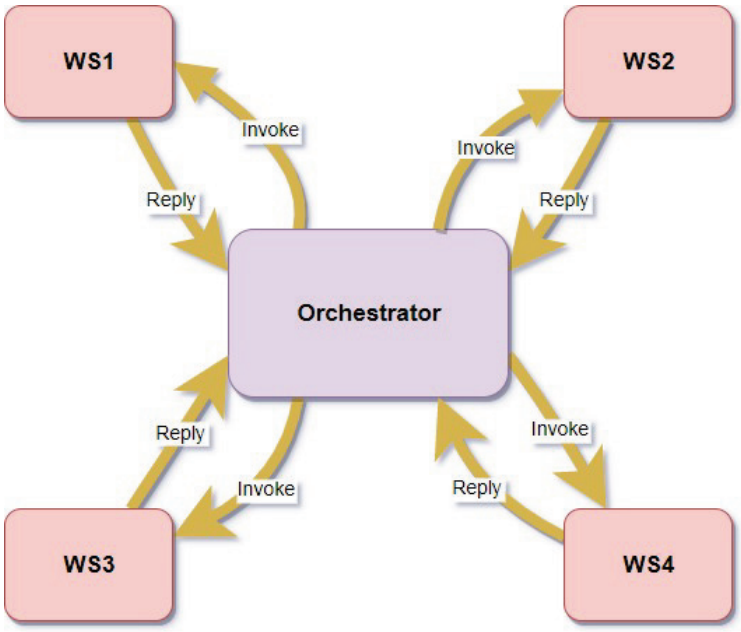

Figure 2.9 - Service orchestration behaviour example

The Orchestrator is aware of all relationships between the participating services, enabling him to act according to the composition specifications and perform the execution workflow, thereby, Service Orchestration enables the system to keep up control over the sequence.

The main support standards which are widely used for design of *Service Orchestration* are Business Process Model and Notation (BPMN) (White 2004), which supplies a graphical representation of the sequence, and Business Process Execution Language (BPEL) (Jordan et al. 2007), which normally is supported by some type of engine in the SOA, since it's based on XML, XML Schema and WSDL, and therefore easily mapped into Service Oriented systems (Rosen 2008).

2.2.4.2. Service Choreography

In *Service Choreography*, there's not an assumed central entity, on opposition to *Orchestration*, but rather a definition of the methodology (time specifications, conversation definition, …) that should be followed by each of the *Services* involved on the complex composition.

Choreography defines a same level collaboration behaviour between involving entities, aiming at an organized interoperability among distributed *Services* without a superior entity in control of the operations (Peltz 2003). In *Choreography*, the flow of interactions is exposed to all the parties involved (Bucchiarone, Melgratti, and Severoni 2007), and the parties follow the defined rules

that enables them to cooperate in the execution of a given process through a particular interface (Jammes et al. 2005). There are two main approaches for *Choreography*: (1) message-based approach, where messages examination allows the recognition of the processes/tasks to be activated. This is very attractive, since specifying the message protocols/contracts (e.g. by usage of ontological models, or Agents technology), is enough for completing the tasks. This mechanism is also supported by the WS-CDL standard (Web Service Choreography Definition Language) (Kavantzas 2004), and often used by B2B (Business to Business) applications (Rosen 2008); (2) work-component-based approach, where definition the individual behaviour of work-components allows to progressively evolve their process based on simple rules: what capabilities need to be completed, or what behaviours are compatible (Rosen 2008).

In more recent years, the extension of BPMN standard into BPMN 2.0 has been gaining wide acceptability, as it begun to support the modelling of *Choreography* (Poizat and Salaün 2012).

Resuming, in *Service Choreography*, the participants must be aware of their role within the current process, by being aware of when, how and with who to react or proactively execute according to a given context. Each participant may be involved with several other participants, while behaving in different role for each interoperation (see Figure 2.10).

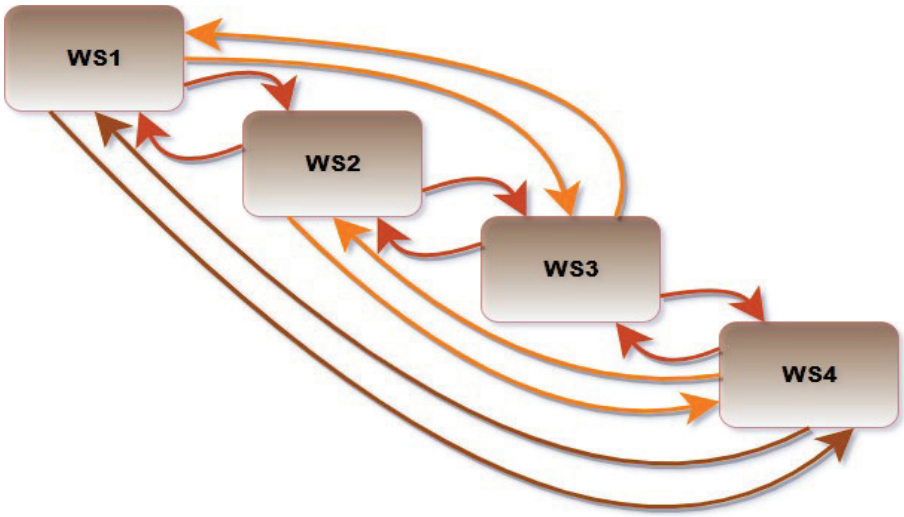

Figure 2.10 - Choreography behaviour example

2.2.5. Multi Agent Systems

Multi Agent Systems (MAS) are referenced in literature as being composed of multiple autonomous *agents*, who dynamically collaborate with each other towards the completion of local and global objectives of the environment (system) they cohabit.

Although there's not a consensus among the several definitions for the term *agent* (Russell and Norvig 1995; Wooldridge 2009), a commonly used one is that an *agent* is an autonomous computational entity that somehow acknowledges (through sensors) the surrounding environment and acts upon it, by performing a decision based on either a pre-determined, learning or knowledge aware logic (effectors).

Also, some widely accepted characteristics are possible to nominate (Camarinha-Matos and Vieira 1999; Monostori, Váncza, and Kumara 2006): autonomy, sociability, rationality, reactivity, proactivity and adaptability. Proactiveness allows the agent to actively seek out its goals without any interference from external entities, basing its decisions purely on its reasoning, its environment and the interactions with other agents in the same community. Since each agent only has a partial knowledge of the environment that surrounds it, the system's goals can only be achieved with the cooperation between the agents that are comprised by a given MAS. In Computer Science, these characteristics make MAS an appropriated solution to be used on open distributed systems, such as the internet (Wooldridge 2009), however, MAS can be found in several applications such process control, manufacturing assembly lines, environment simulation and computer games (Schumacher 2001).

As pointed out in (Cândido 2013), distributed techniques have been separately applied using both SOA and MAS, however they can complement one another, as seen in (Huhns 2002; Luís Ribeiro, Barata, and Mendes 2008). On the work presented on this document, this scenario is applied, as SOA provides the foundations upon which a more complex MAS is embedded, with the functionality of a multi-task and centralized control system, of previously designed and composed services, and that are comprised and distributed on the respective SOA based infrastructure.

3

Overall Architecture & Methodology

As presented on the previous chapter, one possible approach for providing Product Extension Services (PES) is based on the Service Composition Paradigm. In this context, it is considered that the process for providing new PES solutions must be separated in two phases: The Development phase – where the specifications and parameterization of resources are defined, composed and lately compacted into a unique software solution; and the Deployment phase – where the former software solution is then interpreted and made use of, enabling the PES runtime to be prepared and executed.

This chapter aims to depict the overall reference architecture for development and posterior deployment of PES, where both Service Composition and agent-based Service Broker provide the logical functionalities and methodology for enabling the PESs execution. The usage of these concepts implementation, along with the interaction with the rest of the provided components and resources of the infrastructure, in a Service-Oriented environment, provides the further conditions to complete the requirements.

A brief description of the overall modules in use is presented in this chapter: description of their features, functionalities and responsibilities as well as the interactions between them, in order to get an overview of the whole system. However, a more detailed description is made upon the Service Composition Engineering Tool (Development Platform) and of the resources (Services as consumable part for the PES execution), so that finally is possible to give the top highlight to the Service Broker module implementation, as being the main subject of this dissertation.

3.1. ProSEco Concept

The work presented in this dissertation falls under the scope of ProSEco[3] project, which motivation relates to the accomplishment of a novel methodology and comprehensive ICT solution for the collaborative design and deployment of product-services (Meta Product) and production processes. Moreover, ProSEco project intent is to accomplish this through usage of Ambient Intelligence (AmI) technology, lean and eco-design principles and applying Life Cycle Assessment (LCA) techniques allowing effective extensions of products of manufacturers in different sectors (automotive, home appliances, automation equipment, etc..) also bringing enhancement of the product-services and their production processes in the direction of eco-innovation.

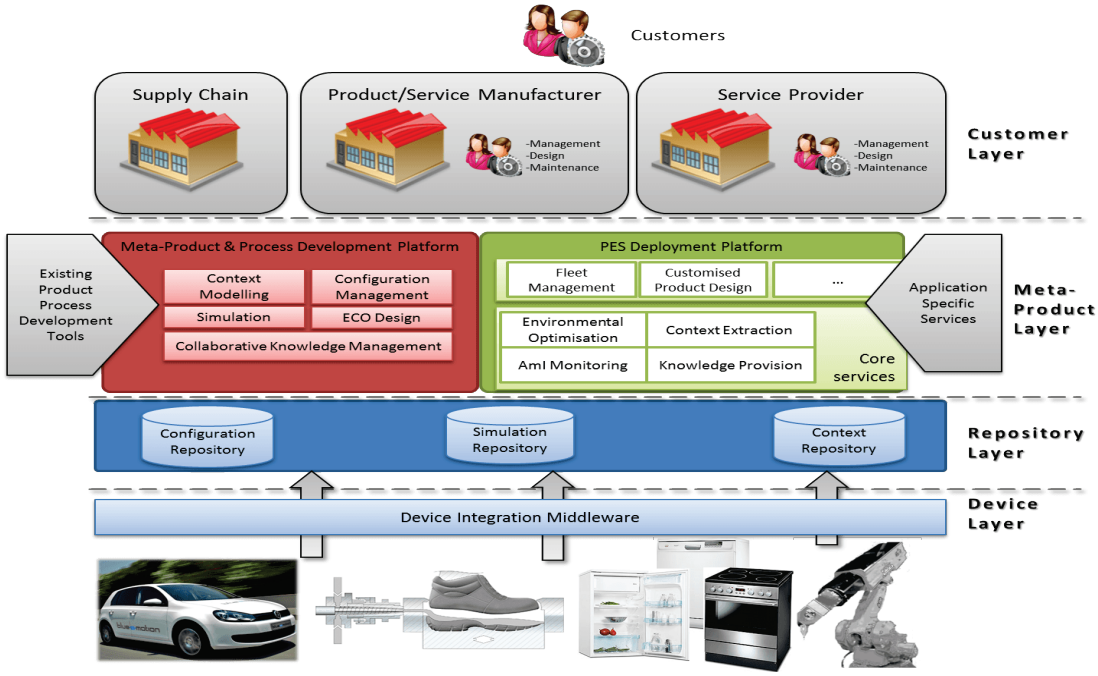

Figure 3.1 - ProSEco Collaborative Environment for design and deploy of PES involving various actors (ProSEco Consortium 2014)

[3] https://www.proseco-project.eu/

In this view, ProSEco stands as a software solution to be integrated and used within an industrial infrastructure. To do so, a multi-layered structure (see Figure 3.1) has been specified allowing the generalization of the several different possible cases that ProSEco is affordable to comply with.

On Table 3.1 a brief description for each of the proposed layers is presented:

Table 3.1 - General Infrastructure Layer Categorization

Layer	Scope
Device	This layer represents the physical machines from where data is extracted: under the ProSEco proejct scope, the concept of Product Service Systems around products is directed to both manufacturing machines and products/processes created or used by them and, as so, this layer represents their physical acknowledgment as well as the direct connectivity to the data provided by them.
Repository	Settled or connected to the corporation or industrial manufacturers infrastructure, the repository layer is where the extracted data from the machinery is stored, and made accessible to be used for the realization of PESs
Meta-Product	This is layer where ProSEco acts, by providing a full infrastructure that enables to perform a methodology for users to achieve the development and execution of PESs. For this, ProSEco offers two highly-reliable platforms that support both design and deployment of PESs, settled in a service-oriented environment, while enhancing integration for newly developed solutions, according to the several paradigms chosen to achieve the desired results, such as: • Product/Process virtualization • Web-based Collaborative design • Service Oriented Architecture on both platforms in use • Specific Application Services driven to the business cases This is to say that this is the level where ProSEco solution will be installed and run, suppling ProSEco users the ability to perform almost all of the methodology that enables to create, design and execute PES
Customer	The entities that consume the results from the executed PES. In here the industrial companies decide to whom the consumable results are supplied, as these can be for internal purposes such as monitoring of the machinery or, on other hand, may serve as actual Extension of Products that may be provided to their customers in some form of business services.

3.1.1. Business cases overview

ProSEco encompasses several application scenarios divided throughout four business cases which, by their turn, are driven by a different industrial partner. A brief description of these business cases and a small extracted set of their associated application scenarios premises are:

- **Business Case 1**: Innovative personalized customer services by exploiting information from AmIs and sensors in the vehicles.
    - Support drivers to optimize energy use
    - Allow Service providers to construct types of services using the information from the vehicles
- **Business Case 2**: Innovative Consumer-oriented products and services considering consumer behaviour and lower environmental impact
    - Provide adaptive control and eco-rating by modelling consumer behaviour
    - Preventive and predictive maintenance by modelling component behaviour
- **Business Case 3**: Implementing improved services for production systems to allow production on demand & optimize maintenance processes
    - Improve man machine interaction
    - Remote diagnostics for support maintenance tasks
- **Business Case 4**: Developing new services for remotely monitoring the supervision of machines and improve the product design process taking customer's preferences and patterns
    - Provide customization in the design of parts by component suppliers
    - Use sensorial information from the machines and apply it on extended services for supporting remote diagnosis and improve maintenance

3.2. Requirements Analysis

Research, industry and academia partners elevated an active joint effort that resulted in the definition of the collection of requirements that ProSEco solution must comprehend for both business cases and general scopes.

The requirements are classified in the categories presented on Table 3.2, based on the proposition given by ISO9126, but extracting the sub-category Security out of Functionality, and turning it into a new category:

Table 3.2 - ProSEco requirements Taxonomy (based on ISO9126)

Requirements Taxonomy	
Category	**Description**
Functionality	Identify functions that satisfy identified needs for the companies
Usability	Identify type of users, degree of expertise and kind of interaction with the systems in order to identify measures that minimize the effort needed for use of the solutions depending on the rank of users
Reliability	Identify requirements of availability of the solution (just if the solution must comply with a level of performance under stated conditions for a stated period of time)
Efficiency	Identify requirements of time of response (just if the solution requires a specific level of performance)
Maintainability	(If needed) identify requirements that minimize the effort needed to make specified modifications
Portability	(If needed) identify requirements that enable the solution to be transferred from one environment to another
Security	Identify limits of access to the solution and information repositories

This collection of requirements refers to the general needs that ProSEco solution has to encompass, however, the overall solution is driven by industrial partners applying several application scenarios. Considering this, a more scrutinized and particularized research over the general requirements was made by analysing the different proposed scenarios.

The applicability of this requirements classification to the general needs that ProSEco Solution must encompass and considering that the overall Solution is driven by several application scenarios from industrial partners, the scrutinized collection of requirements has been elaborated. From this collection, the most relevant for the work presented in this dissertation are:

- Functional Requirements
    - support collection of environmental data, consumer behaviours and data from manufacturing systems for further processing and improvement of Product and Product Extension Services (PES) design
    - provide a collaborative space where the different stakeholders can exchange ideas while designing and building PES

  - provide easy and secure access functionality to the product and services data Knowledge Base (on the cloud) for PES
  - provide infrastructure for deploying various Product Extension Services
  - provide a mean for easy re-configuration (orchestration) of Product Extension Services, which will allow for easy combination of various services and customisation for various customers
  - support building secure services by providing a template for services creation in a form of core service, which can be easily customised for specific application services
  - and in particular, the ProSEco engineering tools shall be a web-based solution
- Non-functional Requirements
  - The following list of key general non-functional requirements was extracted from the analysis made to consortium and extra consortium requirements. The ProSEco solution shall:
  - comply to common dialog design paradigms
  - be user friendly and intuitive, making the utilisation of the solution easy, but without causing distractions
  - provide access to all data, which is monitored by the solution, to the user to achieve user acceptance and trust
  - enable continuous operation
  - fulfil standards and conventions on software design
  - allow for update and maintainability of the system data with minimum effort
  - not affect execution of existing software, which runs in the same software environment
  - respond to user actions in an appropriate time frame

The complete list of general requirements of the ProSEco solution can be consulted on (ProSEco Consortium 2014). These requirements also served as base to elaborate the Business scenarios related requirements.

3.3. **ProSEco Architecture**

From the full collection of the requirements it has been induced the overall ProSEco architecture that satisfies the infrastructure and also the industrial specific needs, while enhancing the system agility and scalability so that not only the system is able to operate in dedication to the application scenarios that were defined for developing and testing the project, but also to make possible to integrate and use new solutions after the full prototype is completed.

From the first version of the defined architecture, several adjustments and upgrades were performed along the development and testing of the solution until a fully functional version was achieved (see Figure 3.2).

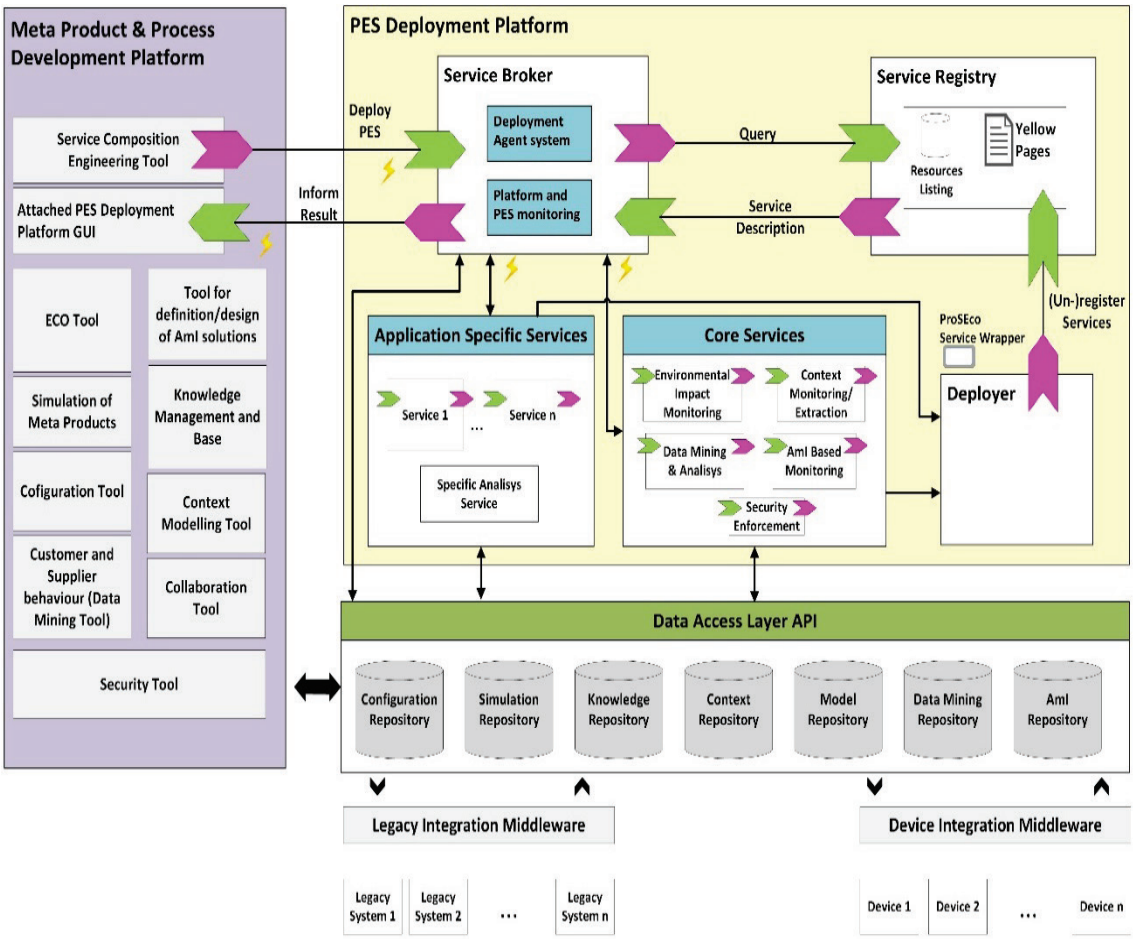

Figure 3.2 - ProSEco Collaborative Environment for PES development and deployment (ProSEco Consortium 2014)

As stated in the beginning of chapter 3, the complete provisioning of PES evolves by two distinct phases – PES Development and PES Deployment. With this in mind, also ProSEco architecture is constituted by two main platforms, each one dedicated to the respective mentioned phase:

- Meta Product & Process Development platform
- PES Deployment platform

Collaboration and interoperability between the components belonging to both platforms are achieved by means of an ontology based system, leveraging a comprehensive workflow along the development and the deployment of the created PES. In this sense, an ontological model was defined to support the implementation of the components of the infrastructure as well as to allow that data inherent to the created PES may be interpreted and used by these components.

It must be considered two stages of the complete workflow for producing new PES:

1. *Development of PES*: Where the users make use of the provided tools supplied by the Meta Product & Process Development platform to create and compose a software data structure that is representative of the constructed PES, which will contain all relevant information that was defined along the PES development.
2. *Deployment of PES*: Where the previous data structure is received (on the PES Deployment platform) and consumed so that it can be executed accordingly with the specifications included on it, in a real-time environment, which by its turn is adopted of autonomous mechanisms that provide such functionality.

Figure 3.3 is elucidative of the straightforward sequence that is necessary to follow for achieving the final results of a PES, by usage of both platforms offered by the ProSEco solution.

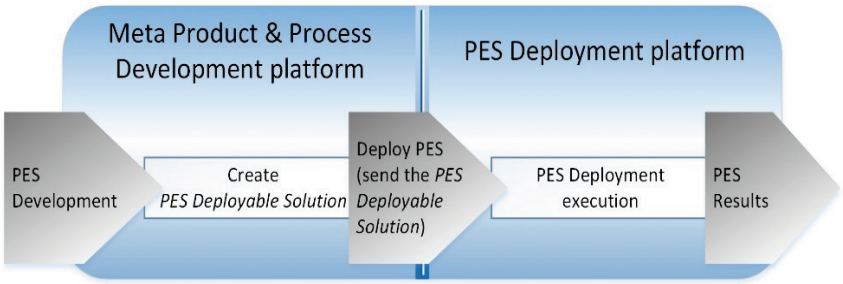

Figure 3.3 - PES Development and Deployment workflow

Baring this in mind, by extending the ProSEco Ontological model,a data structure was defined with the intent of encompass all the information of a PES into a unique object, containing all the attributes, definitions and specifications that were created and produced along the PES development, and that is to be consumed and executed in the PES deployment phase. On Figure 3.4, a simplified representation of the data structure, named *PES Deployable Solution*, used for this purpose is presented.

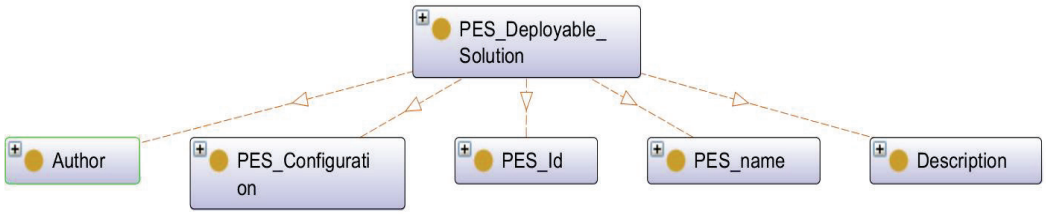

Figure 3.4 - Simplified PES Deployable Solution data structure

3.3.1. Meta Product & Process Development platform

The Meta Product & Process Development platform intents to provide the users to perform the digitization of their products/production processes as well as all necessary mechanisms for the PES designers to configure/design their own PES associated to their respective products. This platform consists of a collaborative environment where the users may access a role of Engineering Tools which enables the digitization of products, the ability to specify and parameterize the resources to be used on the deployment of a PES and finally allows the users to define the *Service Composition* that allows to define the interconnectivity between the resources in use, along the execution of a PES. Each of the Engineering Tools provide an objective functionality related to PESs, but may be isolated from or allied to a specific type, depending if their result is to be (or not) consumed on the deployment environment (PES Deployment phase). The Engineering Tools that not comply directly with the deployment are not relevant for the work presented in this dissertation but, however to highlight the capability of the Meta Product & Process Development platform, the users have at their disposal, for example, a Market Simulation Tool.

On the other side, there are Engineering Tools that may be used to develop deployable PES, that is, the outcomes of them are somehow and somewhere consumed by a certain component belonging to the PES Deployable platform during the PES deployment execution.

The Development environment supplies the following Engineering Tools:

- *Configuration Tool*: Provides the users the mechanism to perform the digitization of the products/processes for which PES may be after developed over. As an example, a manufacturer company may specify its products (model, serial number, type, …) as well as the sensors specifications that belong to each of these products, in compliance with the specified ontology. This enables that other tools may interpret and use this information to create a PES.
- *Deployable Services Engineering Tools*: In general, each service to be used on the PES deployment has an associated Engineering Tool on the side of the Meta Product & Process Development platform, that allows to fulfil the parameterization of the respective *Service* for the runtime execution. Other way to understand this is saying that ProSEco resources are Solutions formed by pairing an Engineering Tool with the respective *Service*, where the first is hosted on the Meta Product & Process Development platform and the latest is consumed on the side of the PES Deployment platform. ProSEco can be used with:
    - *Core Engineering Tools*: These provide the most generic functionalities associated to the respective *Core Services*, such as the AmISelection Tool, Context Modelling Tool or the Data Mining Tool.

- *Application Specific Engineering Tools*: part of the Solutions (Engineering Tool + Application Specific Service) specifically developed and integrated by the manufacturing companies to accomplish objectives oriented to their own specifications, like for example, the connectivity to their internal systems.

- *Service Composition Tool*: Offers the Graphical Environment to PES designers to compose the chosen services of a PES, by defining and parameterizing the data flows between them. Also, it's responsible to aggregate all the information of a PES supplied by the other Engineering Tools into a unique software solution (*PES Deployable Solution*) and to deploy it into the PES Deployable platform, serving as a communication outpoint.

The collaborative environment offered by the Meta Product & Process Development platform support the development of PES by suppling the Engineering Tools that enable the process to create and develop new PES. To support this idea, on Figure 3.5 is presented a generic workflow where Engineering Tools are used in various stages of the development process of a PES, until the deployment is triggered. Information can be passed among the different tools along the process, and finally the Service Composition tool aggregates all the necessary information into a *PES Deployable solution* which is sent to the PES Deployment platform, by means of invoking a service available of the PES Deployable platform that, on his side, serves as entry point (Service Broker endpoint).

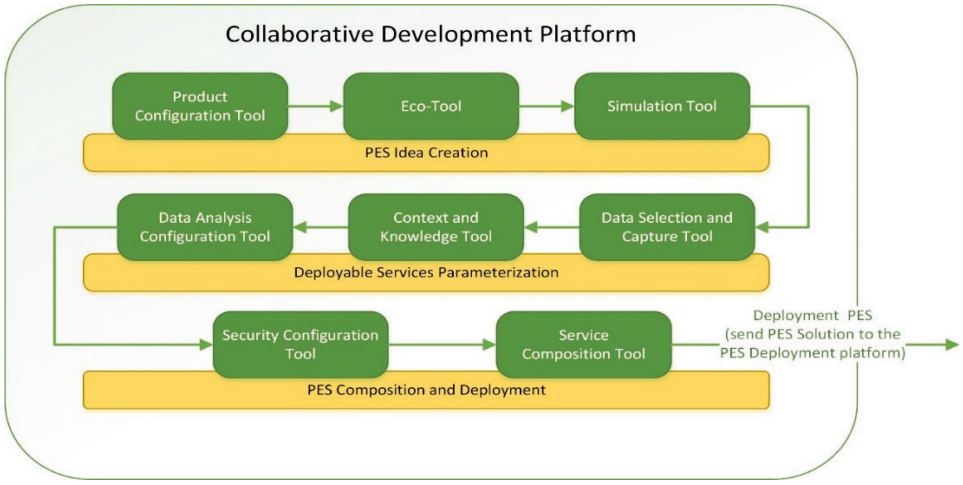

Figure 3.5 - Example of a workflow of collaborative PES development and leveraged Engineering Tools

3.3.1.1. PES Development

It has been defined that the development of a PES that is meant to be executed in the Deployment environment needs to be able to combine all the information that is to be consumed by the resources either for:

1. self-realization of their own functionality and
2. for the performance of the interoperability among one another.

Considering the first, by using the Meta Product & Process Development platform collaborative environment, the PES designer acts by selecting a set of engineering tools (that is, to select the services that are to be part of the PES execution) and then, the PES designer may access to each of the selected tool so that the respective service can be parameterized according to the objective of the PES. On each engineering tool, a configuration file (*PES Configuration*) is created and stored on a repository, keeping its retrieval by other engineering tools available, during all the PES development lifecycle. Also, each of these *PES Configuration* must be passed into the respective service that will consume in order to be prepared to execute the PES accordingly.

For the latest, and according to workflow seen in Figure 3.5, the PES Designer must use the **Service Composition Tool** as the last step of the development phase, since it stands for the exit point of PES from the Meta Product & Process Development platform, and into the PES Deployment platform). Besides that, the Service Composition Tool offers the PES Designers the ability to define and parameterize the data flows (between the internal services) involved on the PES execution (see Figure 3.6). For this, there's an interaction with a knowledge repository where, by usage of the *PES unique identifier* (PES_Id), the *PES Configurations* associated to the PES are retrieved.

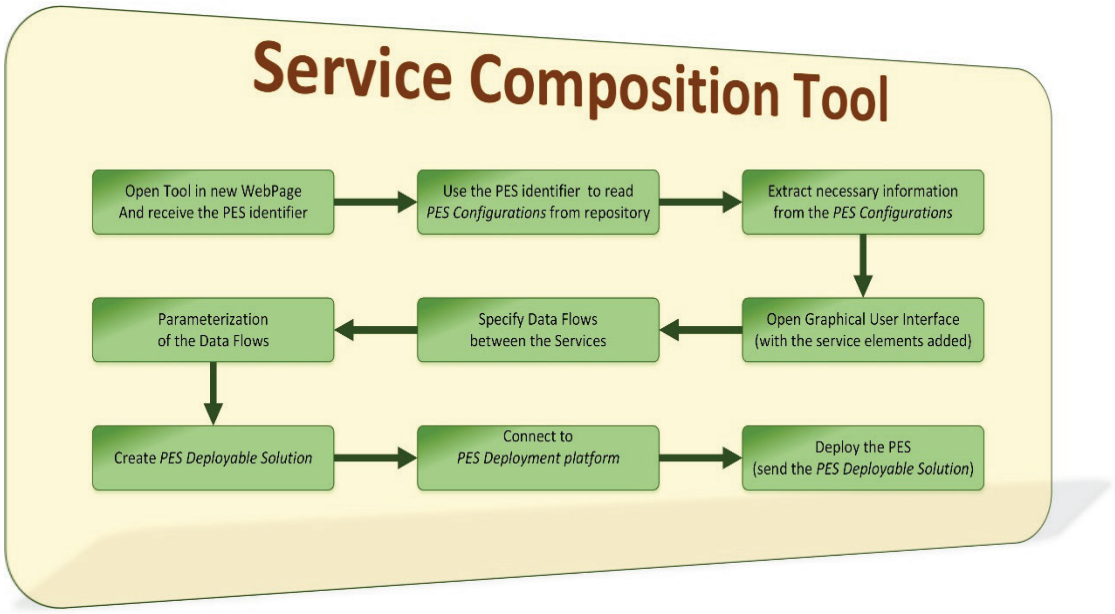

Figure 3.6 - Service Composition Tool workflow

Finally, after the users finishes the data flows specifications, the Service Composition is responsible to gather this information in its own Configuration file (*Service Composition*

Configuration), and construct the *PES Deployable Solution* by joining the all *PES Configurations*, and other relevant information (e.g. creator name, PES name, …). At this point, the user may deploy the PES by sending the constructed *PES Deployable Solution* to the deployment environment (PES Deployment platform).

On Figure 3.7, it's presented the simplified interaction between the Engineering Tools in use (keeping in mind that are destined to parameterize a selected service to be used on the PES execution), the Knowledge repository and the Service Composition Tool, upon the PES Designer(s) intervention along a PES development.

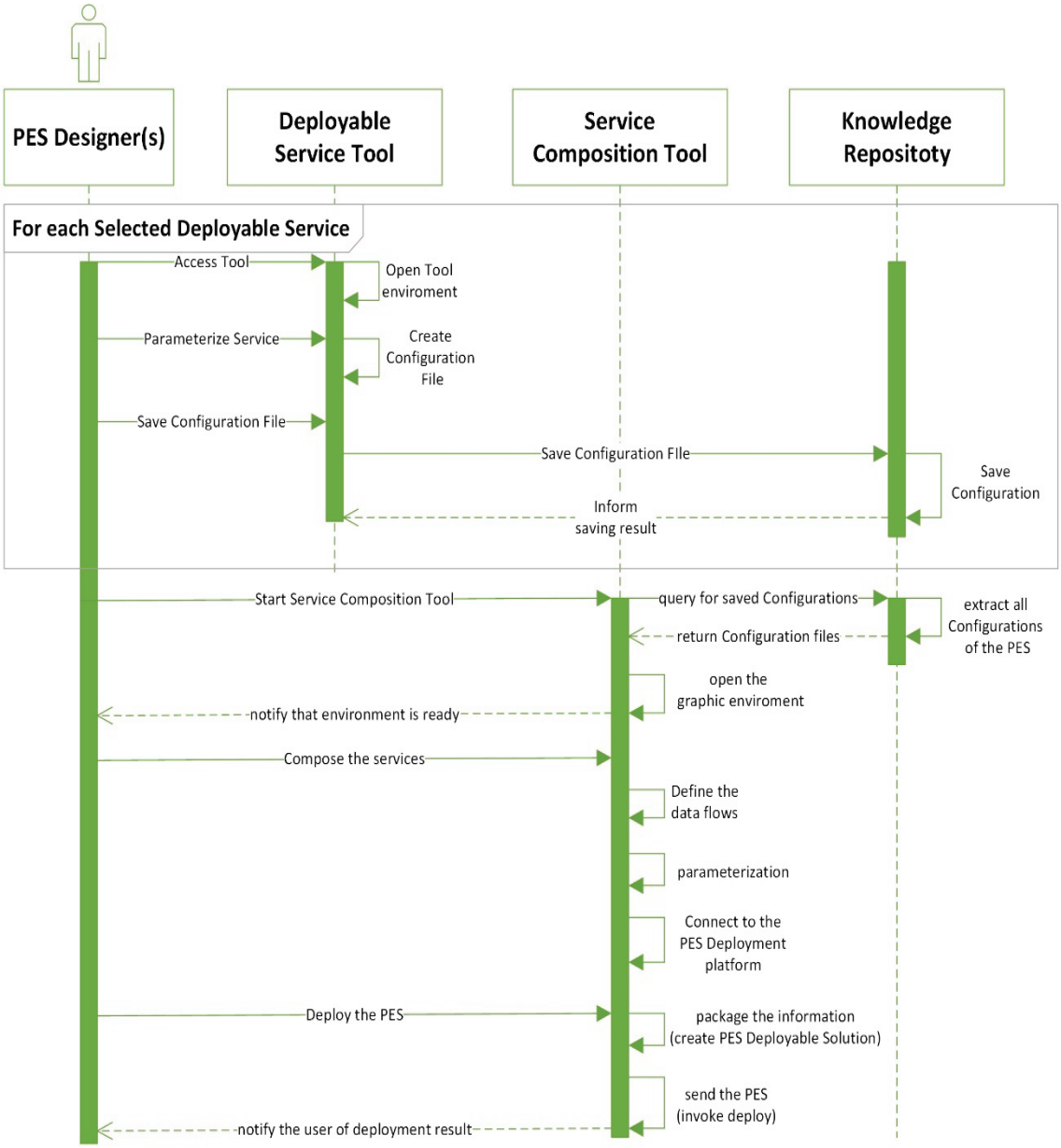

Figure 3.7 - Collaborative development of a PES

3.3.2. PES Deployment platform

The PES Deployment platform provides all the necessary mechanisms that assure the execution of the previously designed PES. The PES Deployment platform is composed by several core modules and functionalities that enable the execution of PES by action and interoperability of these modules, based on a SOA environment. The modules that were implemented are:

- *Deployer:* the system module that is responsible for launching and hosting the resources (as web services) in some machine that is running with ProSEco system, as well as to manage the communication with the *Service Registry* for (un)publish the services, to let the system be aware of their availability.
- *Service Registry:* a database running inside the ProSEco system that allows the discovery and supply information of the availability of resources provided by the Deployers that can be potentially be used in the execution of deployed PES.
- *ProSEco Core & Application Specific Services (Deployable Services):* the several resources that offer atomic and specific functionalities that can be easily embedded and combined to execute PES. As stated in section 3.3.1, each of this resource are part of a combined Solution, where the respective engineering tool that is hosted in the Meta Product & Process Development platform is used to parameterize the usage of the service in dedication to the PES for which it has been selected to operate.
- *ProSEco Repositories:* These are repositories that are associated to a respective *Core/Application Specific Service* in order to store/retrieve the data created during the execution of a PES, and are adopted of the mechanisms to allow the dataflows that were defined to be performed.
- *Service Broker:* an agent-based engine that receives PES Solutions (a PES that is compressed in a software format and sent to the Service Broker Service) and triggers all the mechanism to setup and start the involved components that provide the PES execution. The received *PES Deployable Solutions* that contain all the information regarding the service composition of PES are interpreted and then each resource is accessed in order to send/receive the respective information that enables their setup for the runtime execution.

All the mentioned components of the PES Deployable platform are implemented as web-services, enabling the interoperability amongst each other, according to the methodology in use by the system, and implemented under the structure presented in Figure 3.8, where inheritance plays a crucial feature for providing scalability to the system.

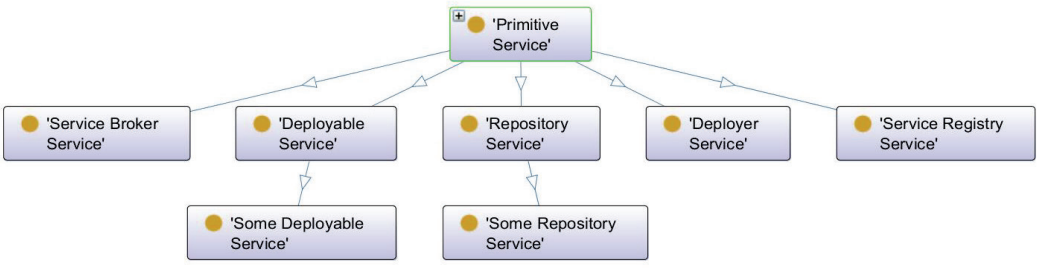

Figure 3.8 - PES Deployable platform components (as services) inheritance structure

The usage of this structure allows that every component may be adopted of generic features (settled on the higher-level classes of the structure), either by the defined generic implementation of the system and by provisioning of placeholders to be complemented during the development of the resource component.

According to Figure 3.8, every service is extended to a *ProSEco Primitive Service*, and by definition, will inherit its methods and values. In this case, the *ProSEco Primitive Service* is an Interface (meaning that, in this case, provides the placeholders for the methods) that supplies four main methods regarding the management and usage of the ProSEco services:

- *Start*
- *Stop*
- *Restart*
- *Ping*

As for different components, since they have non-equal behaviour, each of the placeholders for the above cited methods on each class may differ from one other, such as, for example, the *Start* on the *Service Registry* is used to initialize the registry database and on the *Service Broker* is used to initialize the agent system and its sub-components. These will be seen in more detail on the following sections of this dissertation.

Other important consideration, regarding interoperability between the system components, is that they are provided with an endpoint with the necessary methods and implementation that allows the accomplishment of control and data exchange.

3.3.2.1. Service Registry

The Service Registry stands for a centralized database which aims to provide discovery and availability of the resources (Deployable Services and Repositories) that can potentially be used for executing a PES. In this context, the *Service Registry Service* contains the necessary methods that leverages *Deployers* to (un)register the resources locations, as well as to other components query about the existence, status and availability of these same resources, as they may be acting in dedication to a PES already being executed, which in this case, the resource must not be achievable for other PES, that is to say, the resources have exclusiveness to be acting on one and only one PES at the time. For this, the Service Registry will be updated with the status of any allocated resource every time it is requested.

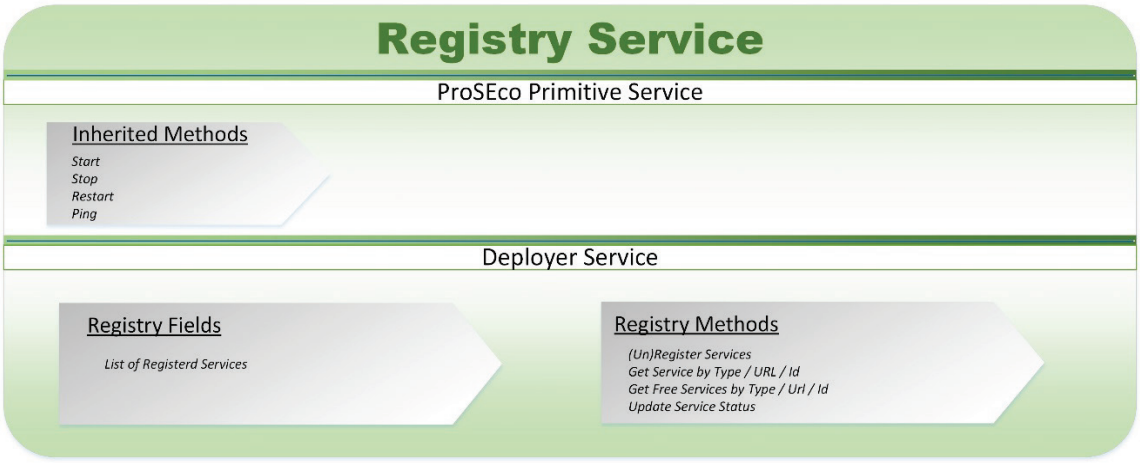

Figure 3.9 - Service Registry Service functional structure

3.3.2.2. Deployer

The *Deployer* is the component responsible for launching the resources (*Deployable Services* and *Repositories Services*) endpoints on the local machine, and further register their information into the Service Registry, by communicating with the *Service Registry Service*. On the other hand, if by any chance the Service Registry occurs into a problem, and if possible, it notifies the Deployer that no longer his services are available on the Registry Database, by invoking the method provided by the *Deployer* endpoint for this effect. For this, the *Deployer Service* has been structured with the following attributes seen on Figure 3.10:

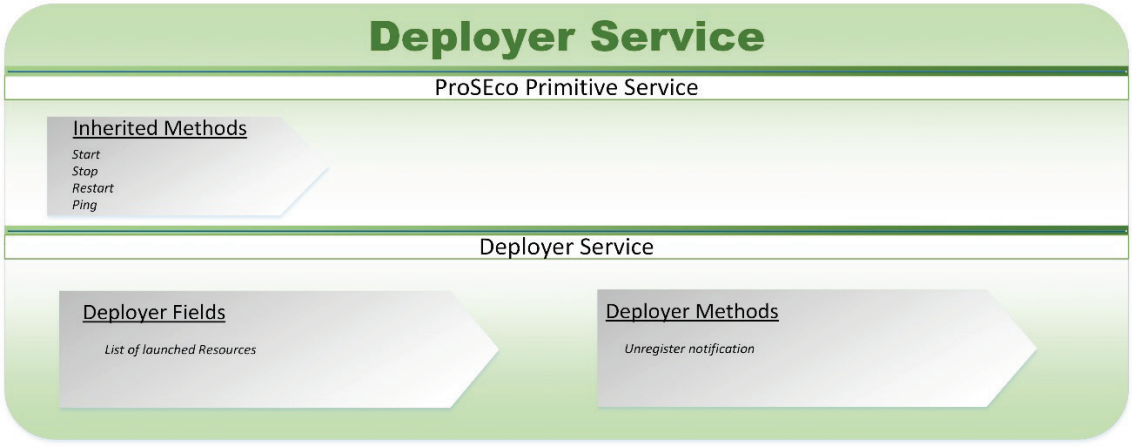

Figure 3.10 - Deployer Service functional structure

The sequential process of the *Deployer* consists on launching the resources selected by the user, update his list of resources and, afterwards, invoke the *Service Registry Service* method where the information regarding the resources supplied by the *Deployer* are received and inserted in the Service Registry database (see Figure 3.11).

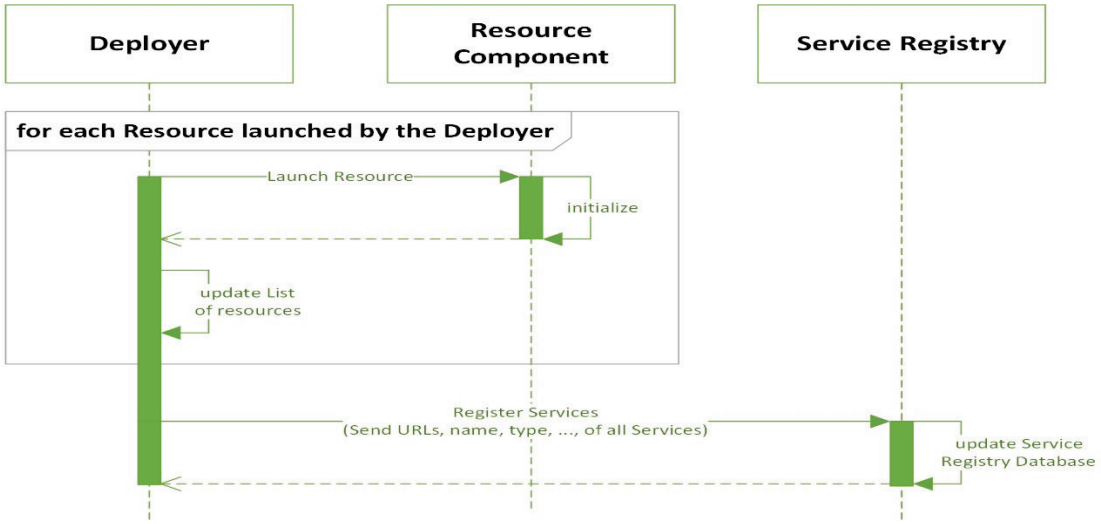

Figure 3.11 - *Deployer* launching the associated resources and registration process

3.3.2.3. Core & Application Specific Services (Deployable Services)

As mentioned in Section 3.3.2, the *Deployable Services* provide atomic (Core Services) and specific (Application Specific Services) functionalities that are used to provide the PES results. Each of these services has its own implementation that guarantees the production of results to be further accessed by other services. Moreover, the integration of these services in the system rely on the methodology provided by the structure of Figure 3.8, where by effect of inheritance, each *Deployable Service* is divided in two complementary implementations: a specific implementation that works to provide the results for which they have been developed and that translates to their functional objective, and a generic implementation, which allows them to be integrated in the ProSEco system and to cope along the fulfilment of the *Service Composition*, and for which they are provided with the necessary mechanisms to be handled along their usage, that is, that allow the system to interact with, enabling their setup and control when a certain PES demands so.

The *Deployable Service* is then provided with three types of generic attributes, beyond the ones already seen from inheritance of *ProSEco Primitive Service,* that are used when preparing and executing a PES received by the system. Their usage allows that a *Deployable Service* allocated to act on the execution of a PES can receive or send information that is required to trigger the interoperability between services, as well as to the system be aware of the status of the PES and of the service itself. The tree types of attributes are:

- *Inherited Fields:* these are variables that are used in dedication to a PES, and in this sense, they either are attributed with values extracted and passed from the *PES Deployable Solution* or that are attributed by the *Deployable Service* itself and are meant to be reached

and passed to another component. As so, these fields will contain information such as the *current PES identifier*, the current *service identifier*, for the first case, and, for example, the *associated Repository URL* in use that needs to be passed to other services that, according to the *Service Composition*, will connect to extract data from.

- *Inherited Methods:* these methods are available as placeholders for developers to implement, but still they are used as part of the generic workflow of the PES execution. As an example, remembering that part of the development of a PES consists on creating configuration files (*PES Configurations*) for each service (see Section 3.3.1.1), that are later to be passed and consumed by the service at this stage, and considering that these *PES Configurations* differ from one another in matter to the respective service, each *Deployable Service* is then capable of receiving the respective *PES Configuration* in the right time of the execution process, but also have the ability to use it in its own terms, through the given implementation provided.
- *Generic Methods:* similar to the Inherited Methods, these are also part of the generic workflow practiced on the setup of the PES prior to the execution starts, but they differ as they have their own generic implementation (see Figure 3.12). They are mainly used to set or receive information to/from the service that will then be used by the *Deployable Service* to perform tasks related to interoperability between it and the other involved Services. An example is the retrieval of the associated repository location that will be passed to other services who are meant to extract data results produced and stored by the service, according to the *Service Composition.*

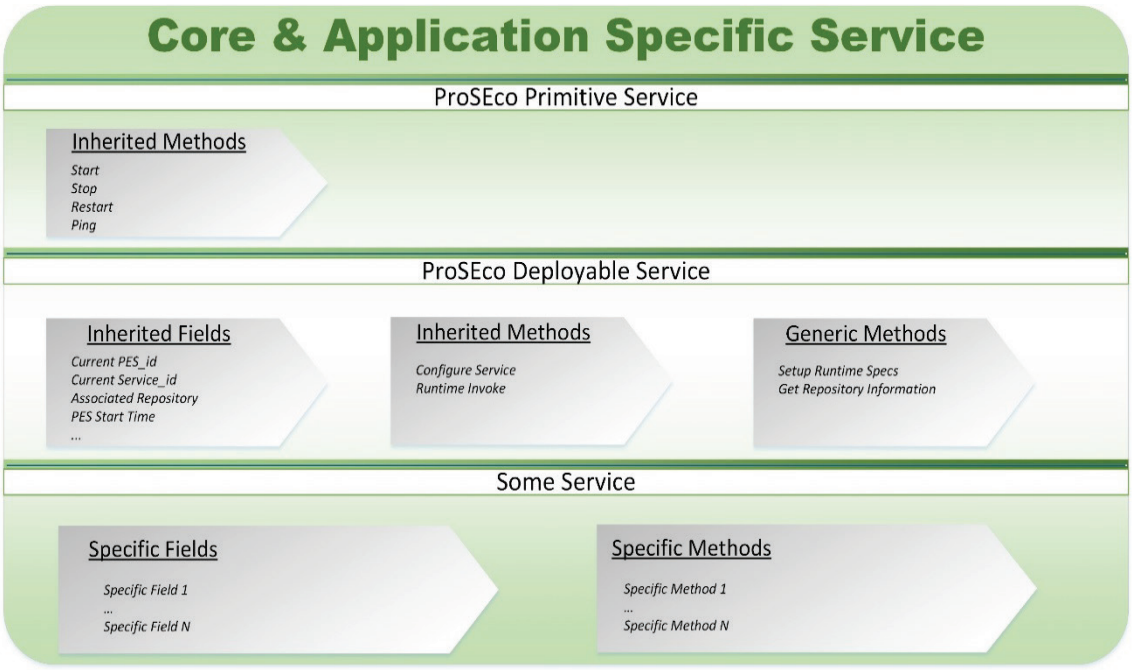

Figure 3.12 - Deployable Services functional structure

3.3.2.4. ProSEco Repository Services

The *ProSEco Repository Services* pretty much follow the same orientation than the *ProSEco Deployable Services* (see Figure 3.13) in the sense that they also possess the same three types of attributes. In the case of the *Repository Services*, regarding the aims for which they are involved in the system, for each attribute it can be stated that are composed by:

- *Inherited Fields:* values like the *OutputDataModel* or the *FlowControllers* that are used or handled along the execution of the PES, leveraging the interoperability of the services.
- *Inherited Methods: Store / Get / Remove data*, as placeholders for allowing the services to have a specific implementation over a generic workflow of the PES execution. The *Deployable Services* to whom the *Repository* is associated uses the *storeData* method to save the data in its own implementation but following some rules in a way that when other *Deployable Service* calls for data, the *getData* method provides the extraction of the data from the repository in the necessary conditions.
- *Generic Methods:* Generic implementation for the setup of the repositories and further execution, in accordance to the *Service Composition* of the PES, such as setting the *OutputDataModel* in use, indication of the *startTime* of the PES or the generic procedure made when other service makes a request for data (*InvokeForData*).

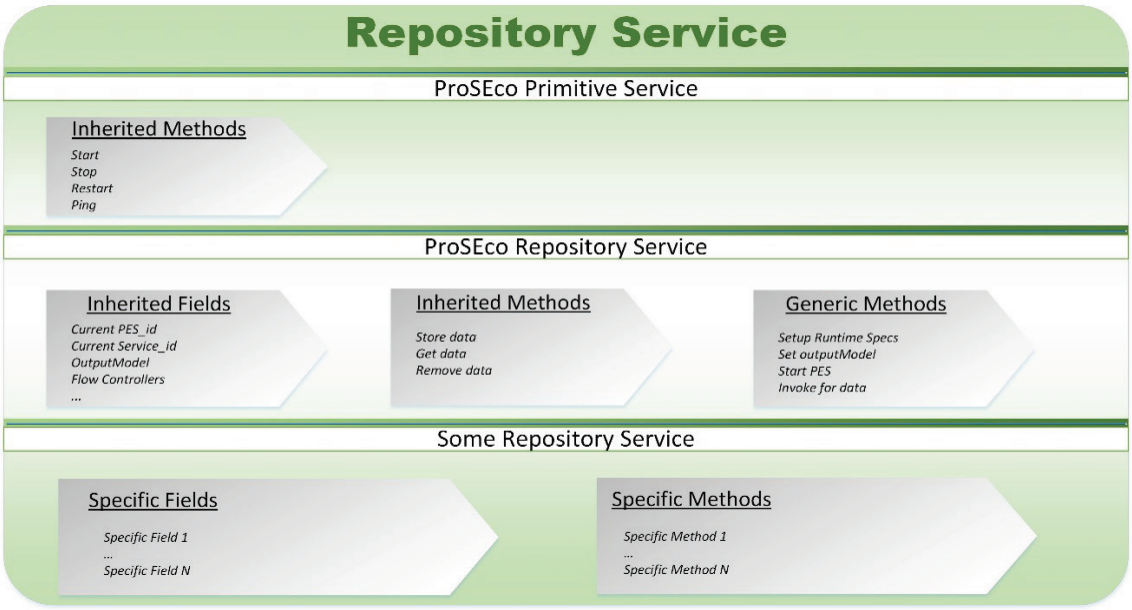

Figure 3.13 - ProSEco Repository Service functional structure

Another important consideration for these components is that they are implemented according to each type of *Deployable Service* they are supposed to be associated with (ex: an *AmIMonitoring Service* works in association with an *AmIMonitoring Repository Service*). This motivates that the

Specific methods implemented on both sides may provide a more objective oriented development regarding their direct involvement, since they can be made only accessible between them.

3.3.2.5. Service Broker

The Service Broker is a multi-functional component of the PES Deployment platform that provides at first hand, the platform's point of access from the outside, in order to receive new PES to be deployed, and also all the mechanisms that enable the PES setup as well as part of the execution. For this, the Service Broker is composed by three interoperable sub-components (see Figure 3.14), which are:

- *Service Broker Service*: the endpoint of the Service Broker that works as the entry point to the PES Deployment platform (as seen in Figure 3.2), serving to receive new PES to be deployed from the *Service Composition Tool*, and commands to stop PES or request to supply information of the status relative to the running PES and the platform itself from the *Service Broker UI*.
- *UI elements:* a data container that keeps all information of the status of the platform and of the running PES, accessible to be retrieved and presented on the *Service Broker UI.*
- *Agent-based system:* the platform engine that acts upon the receival of PES, and is dedicated to the preparation and management of the resources necessary to the execution of the PES, and further keeps monitoring and control of the events that are intrinsic to the PES execution.

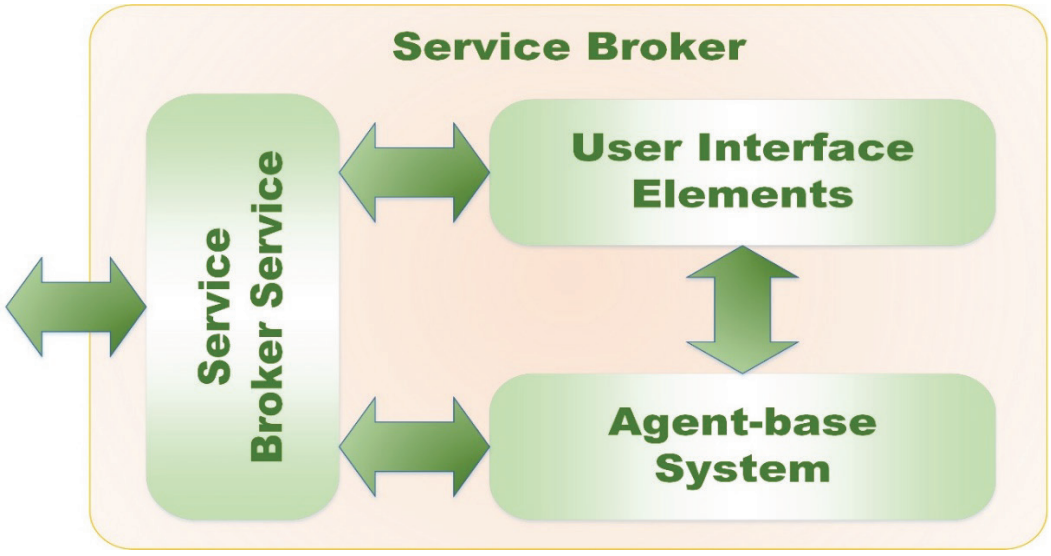

Figure 3.14 - Service Broker architecture

3.3.2.5.1. Service Broker Service

The Service Broker is encompassed on the infrastructure, in similarity to the other components, by providing an Endpoint – *Service Broker Service*.

From a higher-level perspective of the system, the Service Broker provides, through this endpoint, the entry/exit point of the PES Deployment platform enabling the receival of new PES (*PES Deployable Solution*), control request directed to a certain PES execution (e.g. to stop a PES) and request for information (e.g. status of the platform components and status of the PES in execution). For this, the Service Broker Service is provided of the attributes seen on Figure 3.15, which allows fulfils its functional completion for the above stated requirements. In all cases, the Service Broker Service redirects the request to the respective sub-component and replies.

Figure 3.15 - Service Broker Service functional structure

3.3.2.5.2. Service Broker User Interface Elements

Other sub-component part of the Service Broker is the UI Elements. It's an ontology-based data structure (see Figure 3.16) that keeps on being updated while the platform is alive, and aggregates incoming information from the other components of the platform (such as Service Registry) and from the agent-based system (which is dedicated to the PESs execution). The Service Broker UI application (web application) periodically requests to pull this information and further visually presents it to the users.

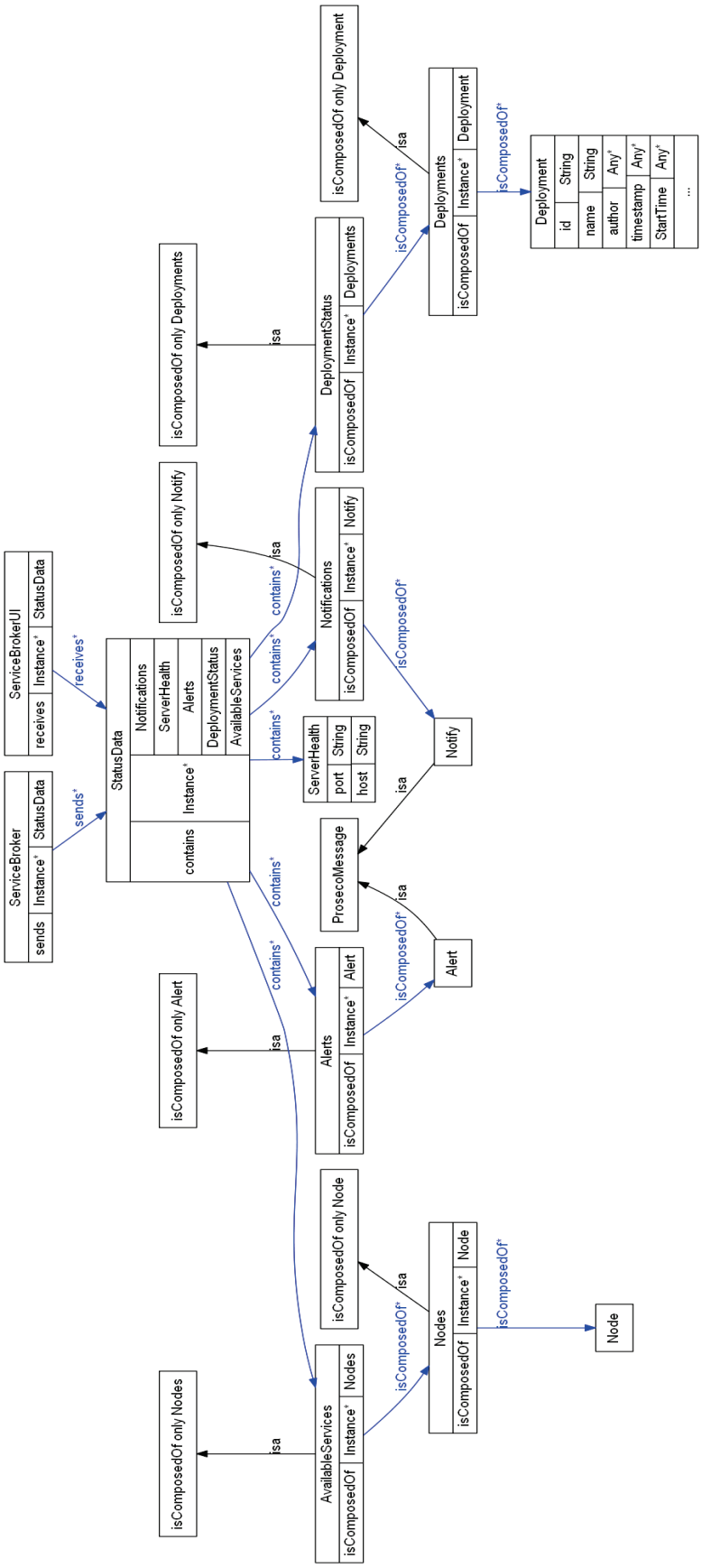

Figure 3.16 - Class relations diagram of the PES Deployment Platform monitoring ontology

3.3.2.5.3. Agent-based System

This component is totally dedicated to the PES deployment, as it provides all the mechanism upon receival of new PES (*PES Deployable Solutions*) that ensure the preparation, setup, control and monitoring of the resources used for executing the PES runtime. For this, the agent system has been adopted with three types of agents*: Broker Handler agent*; *PES Processor agent* and; *Runtime Service agent*, that aim to, in diverse ways, simulate or interact (depending on the case) the relevant actors (see Figure 3.17) part of the system involved on the completion of such tasks:

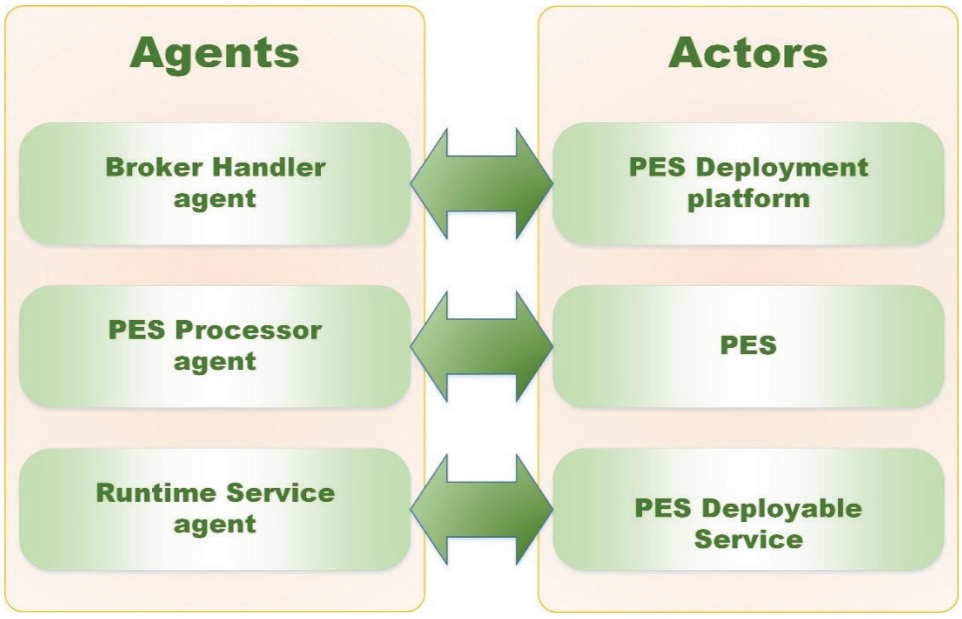

Figure 3.17 - Agent-actor relationship

From the defined relations seen in the previous image, each type of agent has been attributed the following specifications:

- ***Broker Handler agent*:**
  - Since it stands for the layer of the *PES Deployment platform*, there will only exist one instance of this agent running in the system, launched when the system is initialized and destroyed when the system is closed
  - Host and Manage the *Broker Notifications Service* (which description is presented in Section 3.3.3.2.1)
  - Contains relevant information of the Deployment platform, to be passed and used by other agents, such as the *Service Registry Service* location
  - Manages the receival of new PES, and starts the procedures by evaluating pre-conditions for the PES execution (*Service Registry Service* availability, PES already being executed, …)

- Launch and manage the PES Processor agents when a deployed PES passes the pre-conditions
- Evaluates and direct notifications and commands related to the PES in execution (stop command, alert and failure messages, etc.)

Considering the previous, on Figure 3.18 is represented the workflow done by this agent along its lifecycle:

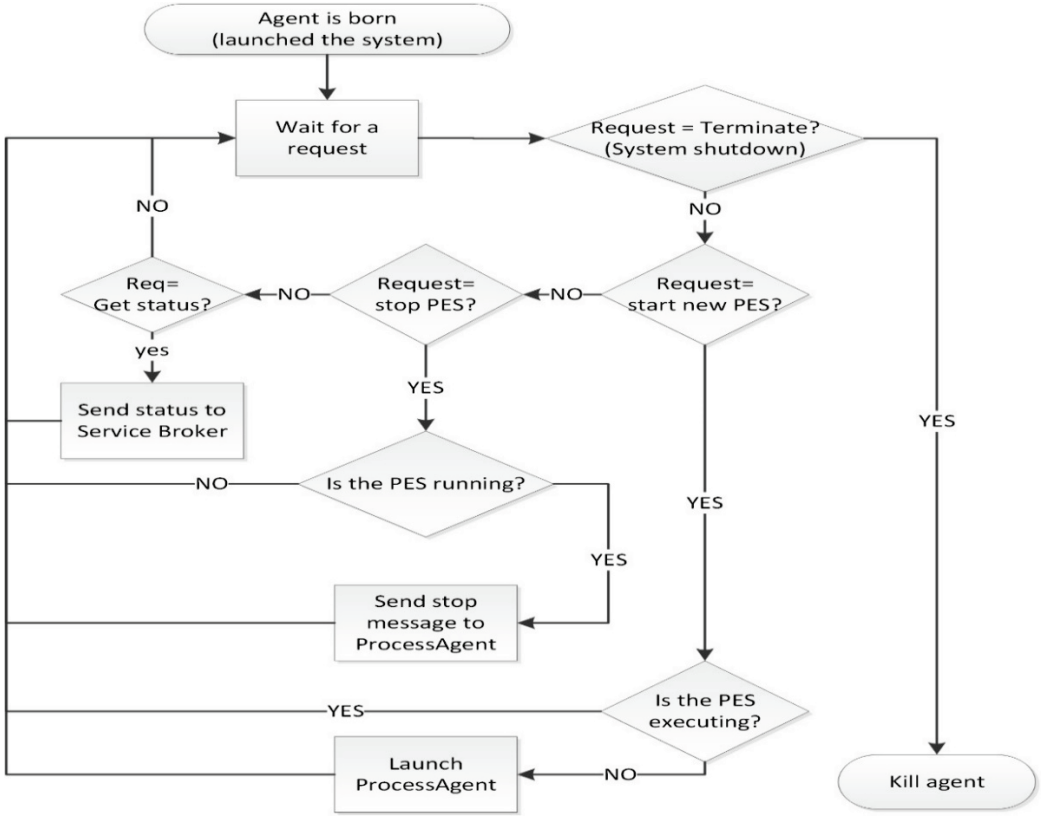

Figure 3.18 - Handler agent workflow

Considering the case that a request to start a new PES is received, the pre-conditions are satisfied and if the PES isn't already being deployed, the Broker handler agent will launch a new *PES Processor agent*, which will receive the *PES Deployable Solution* and will be totally dedicated to the respective PES. For this last-mentioned agent, the specifications are:

- ***PES Processor agent***
  - It's launched by the *Broker Handler agent*, and further receives the *PES Deployable Solution* for which it will exclusively dedicate its works
  - Evaluate the received *PES Deployable Solution* to check for any inconsistency (e.g. there must be one and only one *Service Composition Configuration* and at least one *PES Configuration* from a Deployable Service)

- Retrieve the pre-constructed data structures from the *Service Composition Configuration* that are destined to be used by each *Deployable Service*, and add the respective *PES Configuration*.
- Launch the necessary *Runtime Service agents* (one for each *Deployable Service*), and pass the respective information necessary by them required to function
- React to the incoming messages (request to Stop PES, alert or failure notifications) sent by the *Broker Handler agent*
- Provide synchronization of certain tasks that the *Runtime Service agents* require, by means of agent message protocols

On Figure 3.19, a simplified workflow of the *PES Process agent* is seen, where it is assumed that no errors are found on the consistency check neither on performing the launching and setup of the *Runtime Service agents*:

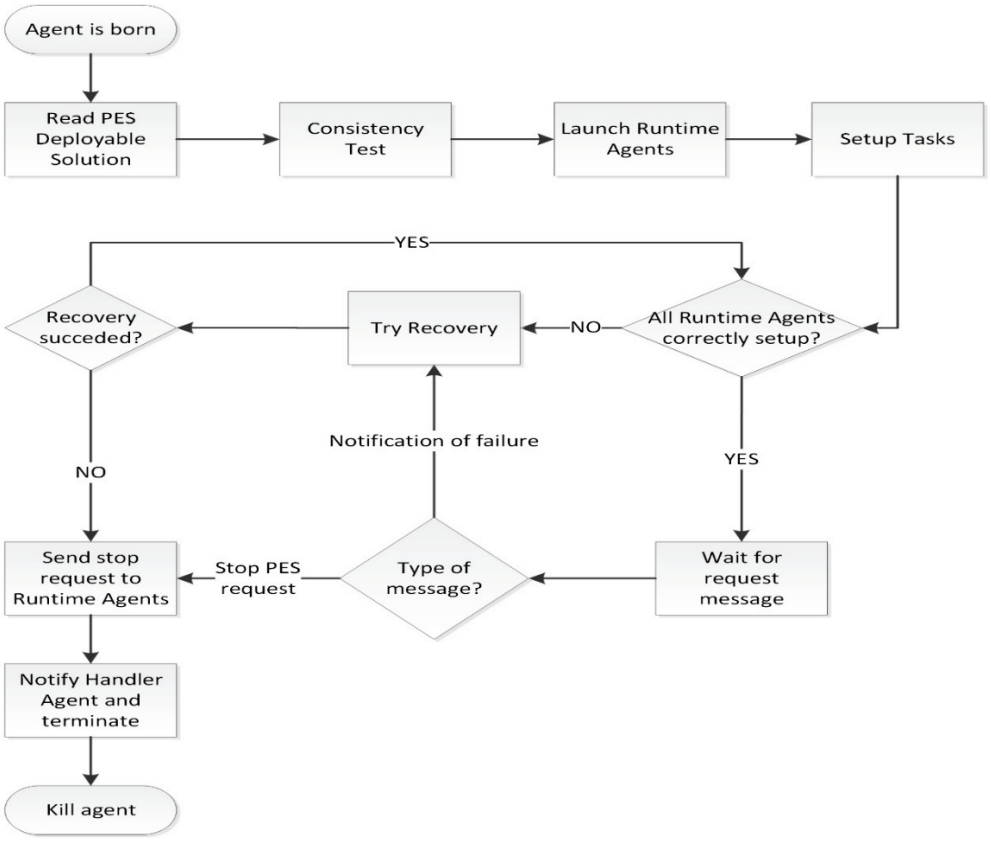

Figure 3.19 - PES Processor agent simplified workflow

As already stated, and deriving from the specifications for the PES Deployment, it is induced that for each *PES Configuration* (that is, for each *PES Deployable Service* supposed to act on the PES execution), a new *Runtime Service agent* in launched, with the following specifications:

- ***Runtime Service agent***
    - Its works are dedicated to a single *PES Deployable Service* to be used on the PES execution
    - The agent is responsible for checking the availability and further allocation (by querying the *Service Registry Service*) of the resource according to its type
    - Interact with the service for obtaining necessary information for the PES execution (the location of the *ProSEco Repository Service* associated to the allocated service)
    - Setup both *Deployable Service* and associated *Repository Service* – pass all the information regarding the system (e.g. *Broker Notification Service* location) and data flow specifications containing the specified times and other Repositories locations, as well as the corresponding *Output Data models.*
    - Configure the Service, by sending the respective *PES Configuration* file
    - Start the service, indicating the start time of the execution
    - Give specific commands to the service, for example, if requested to extract data at specified times or when the request to stop the PES is activated.

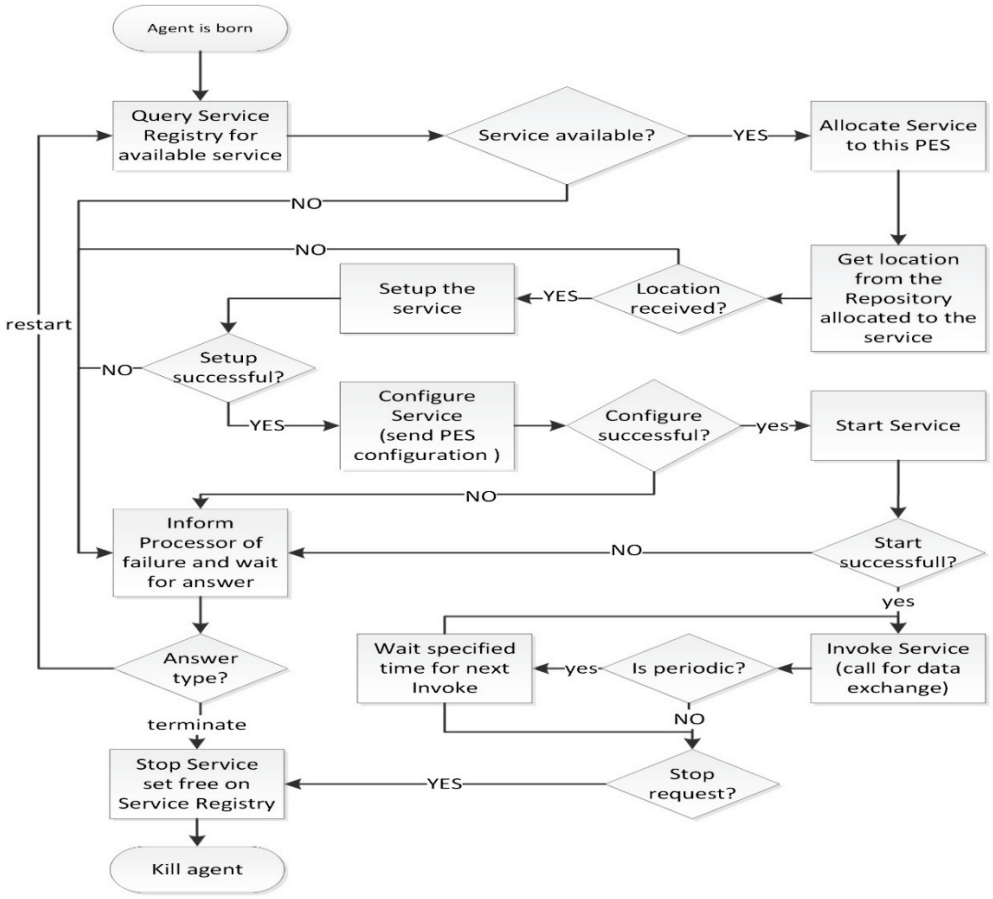

Figure 3.20 - Runtime Service agent workflow

3.3.3. PES Runtime Setup and Execution

In the previous sections the several components that provide the overall environment where PES can be deployed and executed was presented, providing a better understanding of the subject in discussion in the current section. As seen, once a PES is deployed into the *PES Deployment platform*, the agent-system takes over the actions that, according to the definitions intrinsic to the design of each PES, will perform the allocation and setup of the resources that, by their turn, will proceed to the execution. Following this line of thinking, it can be said that the deployment of a PES is divided in two stages:

- *Validation & Setup stage:* where the agent-system of the *Service Broker* has a more active role, by performing the allocation of resources to be used, as well as their setup and establishing the communication conditions between them.
- *Execution stage:* where resources (core / application specific services) in use perform most of the work, while the *Service Broker* mainly serves as monitor of the execution.

3.3.3.1. Validation & Setup Stage

3.3.3.1.1. Agent-based workflow

Once a new PES is received, the system activates the mechanisms offered by the agent-system following the established rules that enable the setup process (see Figure 3.21).

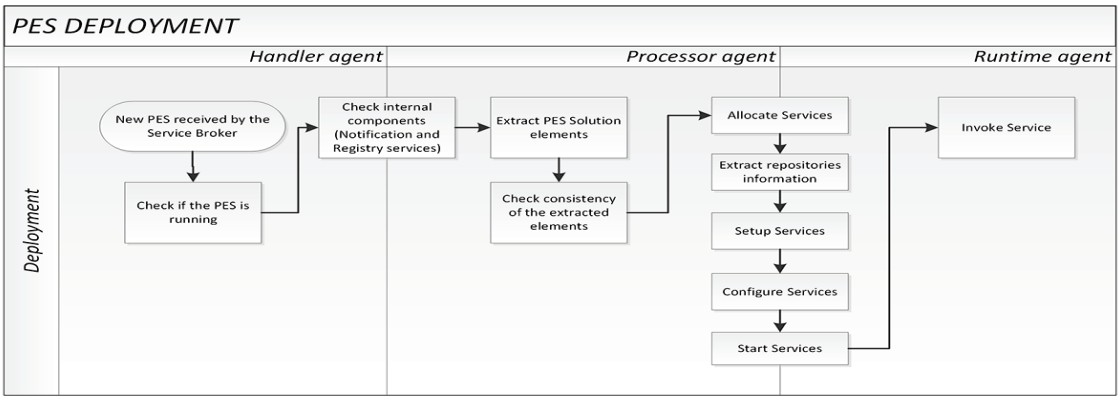

Figure 3.21 - Agent-system simplified workflow for the setup phase of PES Deployment

According to its evolution, agent decisions are made from either self-behaviour or from interaction with other agents, which may include launching new instances of agents (or dispose them). The expected target is to provide the full setup of the components involved on the PES execution following the workflow presented, where the stages that stand between agent columns are representative of tasks that require interaction between the respective agents, along the

workflow. Furthermore, and accordance to what has been seen in Section 3.3.2.5.3, it can be noticed a hierarchical structure of the agent system, despite agents behave as autonomous entities.

3.3.3.1.2. PES Deployable Service and Repository methodology

In accordance to the methodology presented for the agent-based system, both *PES Deployable* and *Repository Services* also follow their own generic methodology for the *Validation & Setup stage.* For this, they make use of the properties referred on Sections 3.3.2.3 and 3.3.2.4, leveraging to the most the generic workflow while leaving space for the developers to implement the specific aspects of the resources.

Focusing on the *PES Deployable Service*, once it has been allocated for a certain PES, it needs to complete a sequence of routines in order to be prepared for the PES Execution (see Figure 3.22), each one triggered by the agent-system, and has seen in section 3.3.3.1.1. The routines implemented are:

1. *getReposInfo*: the service must have a reference to an available *ProSEco Repository* to use for storage of the produce results, and further retrieval by other services. The location of such *Repository*, must be contained on the reply to the agent-system in order for it to include on the data flow specifications that depend on it;
2. *setRuntimeSpecs*: The service receives the *Broker Notifications Service* location, the data flow specifications (containing the *ProSEco Repositories* locations that will be used on data transfers, as well as the time specifications and inherent data models);
3. *ConfigureService (placeholder)*: the service receives the respective *PES Configuration* and uses it according to the implementation provided on resource, replying with a notice of success of the operation;
4. *Start (placeholder)*: the service receives the start time of the PES Execution. The implementation made for this method may be used to initialize some parameters, or it can also be empty, according to the own resource needs.

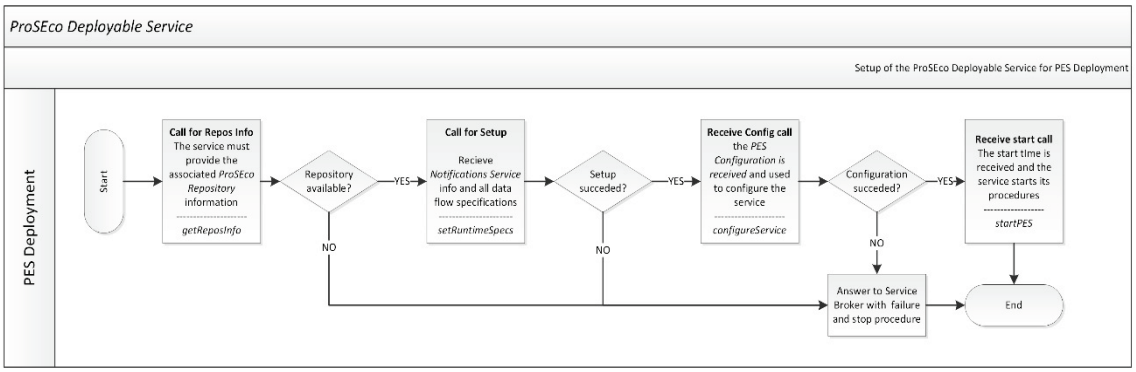

Figure 3.22 - PES Deployable Service workflow on PES Setup stage

On the other side, the *ProSEco Repository services* setup is done in two steps, during the *Validation & Setup stage* which are, in similarity to the previous case, triggered by the agent-system as well (see Figure 3.23):

1. *SetupRepository*: when this method is invoked the *Repository* receives the *Broker Notification Service* location and the data flows specifications (identifiers of the connected resources, data models is use for each defined flow, etc…). The data flows are used to setup the *Flow Controllers* that will be used on the PES Execution.
2. *StartPES*: the *Repository* is informed on the PES Execution start, while acknowledging the respective start time, which is settled on the *Flow Controllers*.

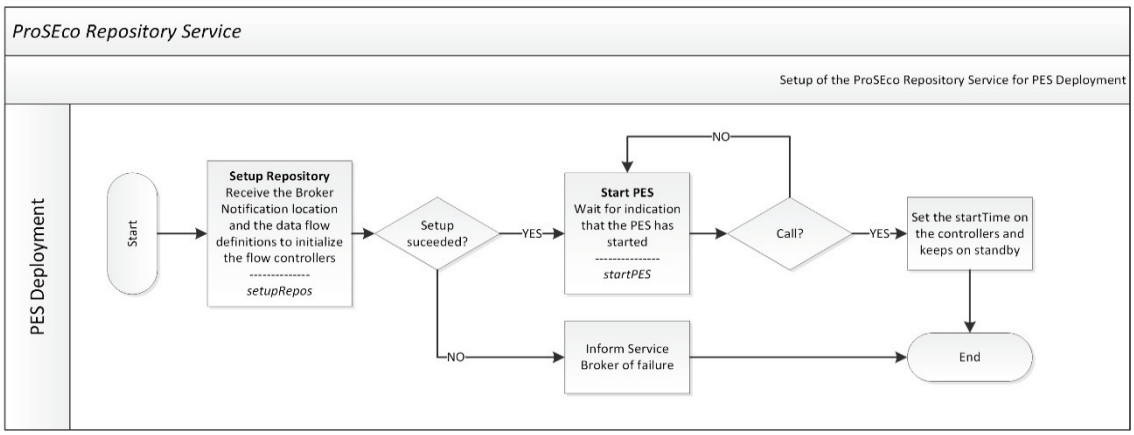

Figure 3.23 - Repository Service workflow on PES Setup stage

3.3.3.2. PES Execution Stage

Once the PES Execution is started, the resources in use (*PES Deployable Services* and *Repositories Services*) that were specifically allocated and prepared for the PES, provide the main role of the Execution, acting in accordance to their objective by using the specific implementation developed, and in compliance with the designed *Service Composition*, by using the generic features that leverages the data exchange of the produced results along the runtime.

On the other hand, the agent-system will keep on monitoring the PES Execution, by keeping updating the status of the platform and of the resources. Since the resources are acting independently from the agent-system, it was found the need to incorporate the functionality that enables them to communicate back to the system. For this, an internal Endpoint has been implemented: *Broker Notification Service.*

3.3.3.2.1. Broker Notification Service

The aim of the *Broker Notification Service* is to allow the resources being used in a PES Execution to send information back to the system, in regard to current evolution of the PES, and in contrast

with the *Service Broker Service* that is directed to outer connections. With this in mind, it should be noticed that on the *Validation & Setup stage* every resource receives the *Notification Endpoint* location so that they are able to connect.

The method provided by this service, named *Notify_Broker*, when invoked, contains the parameter *Broker Notification* which includes several fields in order to be possible to identify the sender of the notification, the associated PES identifier, the data flow identifier and the message to be acknowledged.

The messages are composed relying on an specific ontology, assisting on the interpretation of the type of message and on forwarding the system to take some decision if needed (for example if an error message is received, it can lead the system to force the PES Execution to stop or try a recovery). The notifications can be of the following types:

- *Information*: for example, to inform that a given operation succeeded or a status changed;
- *Command*: can be used to tell the system to take some action, for instance, to stop the PES execution
- *Alerts*: To let the system know of something that has not functioned properly, regarding the PES execution or if an internal error has occurred

To assist on the construction of the *Broker Notifications,* both *PES Deployable Services* and *Repository Services* are adopted with template methods to be used automatically on the generic methods and to facilitate their integration on the specific methods of the services.

3.3.3.2.2. Resources interoperability methodology

From the overall concepts and methodologies described along this chapter, it can be concluded that the resources allocated for the execution of a PES are aimed to fulfil two objectives:

- *Specific functionality*: where the service executes its atomic/specific functionality according to the implementation, providing outcomes that are kept in storage for being used by other services, and;
- *Interoperability functionality:* the generic implementation that must follow up the designed *Service Composition,* which provides the data extraction of the stored results from one given service, by other in accordance to the data flow specifications

On every PES, each specified data flow is relative to two services: the *Sender* and the *Receiver*. Two services involved in a data exchange, from the point of view of the PES Execution and

regarding their *Specific functionality*, act by cyclically (may be periodically or not, depending on their self-implementation) producing results and storing them on the associated *Repository Service*. Then, at any given time the *Receiver Service* may request the associated *Repository Service* of the *Sender* to pull data from it. At this point, as all resources have been already configured during the *Validations & Setup stage*, the associated *Repository Service* of the *Sender* already is aware that anytime will receive this request, and knows how to act accordingly. The *Repository Service* that is requested to supply data makes use of the invocation parameters (*flow identifier*, *PES identifier* and *Service identifier*), to gather the respective data that is assigned according to these same parameters.

The data is collected and transformed into an object capable of being recognized on the *Receiver* service, by use of the specified data models defined during the PES development, and afterwards sent on the reply of the invoked method (*call for data*), as seen in Figure 3.24:

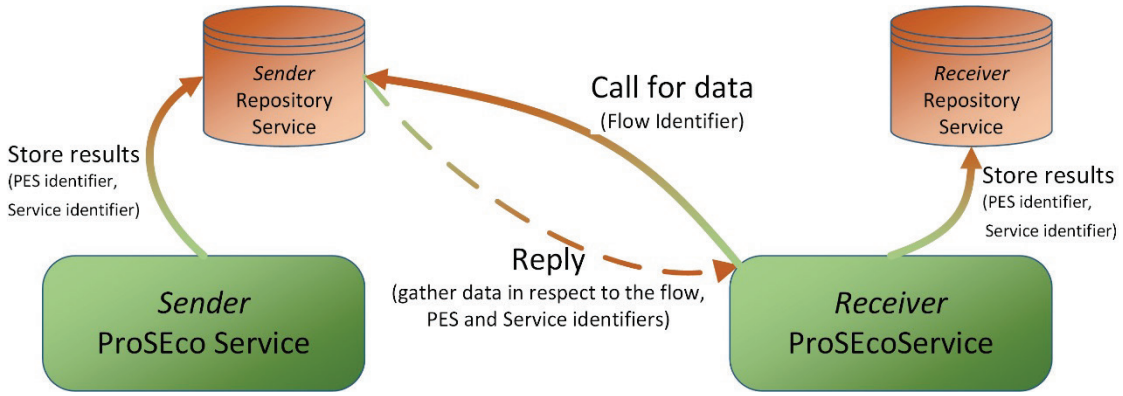

Figure 3.24 - Conceptual model of the communication mechanism between two services

4

Prototype Implementation

This chapter aims to discuss the aspects related to the implementation of components of the overall architecture that have a more strictly involvement on the PES Deployment: *Service Composition Engineering Tool*, *Service Broker* and the generic features of the resources (*PES Deployable Services* and *ProSEco Repositories*), while detailing all the development tools used. According to the guidelines depicted in chapter 3, the resultant prototype was applied and tested by all four industrial partners involved in ProSEco project, where real products and processes encompassed in the specified technical environments were used to validate the concept, feasibility and reliability of the whole ProSEco infrastructure for developing and deploying new PES. Therefore, the application scenarios of each industrial partner's Business Cases served a crucial aid to the implementation, as the several tests performed along provided the base for facing the problems that emerged, and which feedback was used for making the necessary adjustments: handling both specific and generic features of the resources, take the most advantage of the Service Oriented paradigm, enabling the configuration and use of the system over a distributed

network, defining the required parameters to allow the system to interpret the specifications of the Service Composition and perform the execution accordingly, and others, with all contributing to an efficient environment where PESs can be rightfully deployed and accomplish the proposed objectives. In this chapter, generic and/or exemplificative PES are used to overview the implementation process, as they fairly are representative of the real PES which were implemented for the industrial partners test cases.

4.1. Development Software Tools

For the implementation of the main components of the ProSEco architecture, several development tools and Integrated Development Environment (IDE) have been used, assisted by the use of software technologies, most of them well documented and open-source, leveraging the easy integration and combination of their functionalities. In the cases of the *Service Composition* Engineering Tool and of the PES Deployment platform elements, which this dissertation is oriented to, the open-source NetBeans IDE has been used for the development and structure of the elements.

As first consideration, since the development of ProSEco project is a joint effort between the consortium's members, Apache Maven and Subversion Tools were employed. The first provides the standardized management of the project modules, while allowing to build them into the executable JARs, so that they can be shared by several projects. The later one serves to control the versioning of the current implementation, while maintaining track of all the updates, aiding to the revision and detection of conflicts that may occur, as several entities may perform updates on the same sections of the implementation, which may induce to rollbacks or merging of such excerpts of the code files.

Whole mentioned components of ProSEco Solution rely on Apache CXF framework as the foundation for SOA modules, which provides web-service based methods and transport channels, leveraging the development and communication between services. Apache CXF is an open-source framework, that supplies the usage of Application Programming Interfaces (API's), such as JAX-WS or JAX-RS, for the development of web services with a wide possibility of messaging and transport protocols to be chosen (e.g. SOAP, REST, HTTP, …).

The *Service Composition* Engineering Tool, as a module part of the *PES Development platform*, is implemented as a Web Application based on JavaScript, with integrated Bootstrap library for providing a better frontend experience. As for the back-end, the tool consists of a Servlet (ServletCXF) – Server-side implementation for responding and messaging to the service calls integrated in the workflow of the users-, through REST implementation of the service oriented

logic of the tool, aided by JQuery Ajax library for easing the client/server communication. Looking more in the direction of the *Service Composition* functionality provisioning, once it was specified that BPMN standard was to be used, the open-source *bpmn-js* (bpmn.io) library was integrated, while providing the chance to be customized or extended according to the needs of ProSEco. Finally, for both testing and for integration on the *PES Development Platform*, the *Service Composition* Engineering Tool was deployed on Apache TOMCAT server.

On the other hand, all modules of the *PES Deployment Platform* were implemented in Java, as they were envisioned to be employed with the same design and development environment on the prototype, which by its turn it's delivered as a whole Java application (a unique executable JAR that encompasses the specific JARs of each module in use on the application, as well as every other included library). The overall interoperability between modules is provided through the inclusion of services related to each component, relying on SOAP protocol working over HTTP, and implemented with JAX-WS API.

The application visual environment was constructed using SwingX (extension of Java Swing GUI toolkit) elements and JGoodies for the specifying their layout.

The *ProSEco Repositories* have independent implementations according to the developers of the Core/Application Services, meaning that as a Service of ProSEco internal workflow, they obey the rules of implementation, but as repositories they are free to be implemented in any terms the developers desire. Either way, for testing/evaluation of the prototype development and for the services specific development, different databases engines were employed, such as H2DB or MongoDB, with no constrains regarding the associated data base technology (MySQL, NoSQL, …) as long as their available to be used with Java.

The agent-based system of the Service Broker module that provides the PES Setup and Execution is implemented with JADE (Java Agent Development Framework), where several other agent related technologies and protocols are employed, such as ACL (Agent Communication Language) messaging and FIPA (Foundation for Intelligent Physical Agents) protocols, for provisioning a more complete achievement of the proposed goals.

Finally, in the development process and implementation of ProSEco prototype, when applicable, several other development tools/technologies have been used, like for example: Protégé for specifying the Ontological Model of ProSEco System; XML is widely used in ProSEco as standard for data description handling, either manually used for reading and writing local configuration files through simple-xml framework or integrated in other frameworks in use, as in the case of data binding performed by APACHE CXF, where JAX-B (Java Architecture for XML

Binding) is incorporated. Also, JSON (JavaScript Object Notation) is used for Serialization of data in both REST and SOAP services, providing data transactions less susceptible to flaws.

The more relevant Development Tools used, together with the respective functionality and link are listed in Table 4.1.

Table 4.1 - Overview of used Development Tools

Requirements Taxonomy		
Functionality	**Software**	**Link**
IDE	NetBeans	https://netbeans.org/
Programming Languages	Java Java Script	https://www.java.com/ https://www.javascript.com/
Uniform Building System	Apache Maven	https://maven.apache.org/
Version Control	Apache Subversion	https://subversion.apache.org/
Web Application Framework	Apache CXF	http://cxf.apache.org/
GUI toolkits	SwingX JGoodies	https://www.oracle.com/ http://www.jgoodies.com/
Agent framework	JADE	http://jade.tilab.com/
Spring framework	Spring	https://projects.spring.io/spring-framework/
Data Base framework	H2DB MongoDB	http://www.h2database.com https://www.mongodb.com/
Web Application Server	Apache TOMCAT	http://tomcat.apache.org/
Ontology Editor framework	Protégé	http://protege.stanford.edu/

4.2. **Prototype Development**

ProSEco project provides a novel methodology and comprehensive ICT solution for the collaborative design and deployment of product-services (Meta Product) and production processes, based on a SOA infrastructure composed by two platforms, one dedicated to the development and the other to the deployment of PES.

This section details the implementation of ProSEco prototype focusing on the components more strictly related to the *Service Composition* and real-time PES execution: The *Service Composition*

Engineering Tool, *Service Broker* and generic features of the resources (*PES Deployable Services* and *ProSEco Repositories*) in use and of the system itself.

4.2.1. Generic data structures

In order to leverage the development and deployment processes, integration process and interoperability between the components of ProSEco system according to the requirements, the implementation must follow some common aspects presented in chapter 3: working over the accorded *ProSEco Ontological Model* or extending classes to the fitting generic interfaces are examples of this, as it will be viewed along this section.

4.2.1.1. PES Deployable Solution

It was deduced the necessity of creating a data structure able to encapsulate all the features, and specifications of a PES into a unique software solution – *PES Deployable Solution* -, facilitating not only sending the PES from the *PES Development Platform* into the *PES Deployment Platform* but also the comprehension on reading and interpreting the PES, from the side of the *PES Deployment Platform* components that are responsible for providing the setup and execution of the PES.

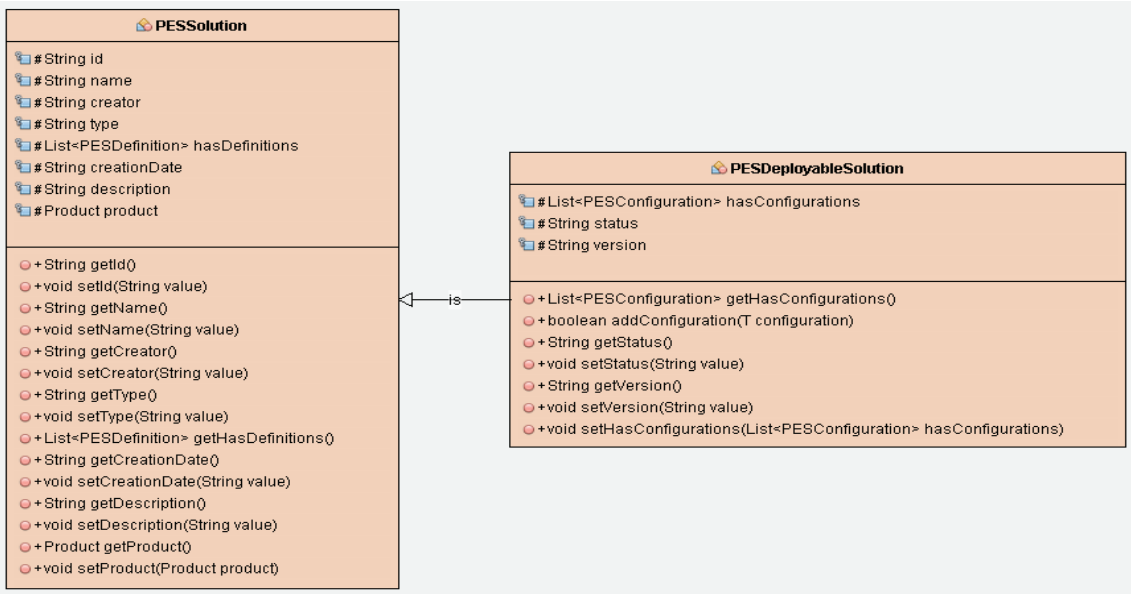

Figure 4.1 - Pes Deployable Solution (and parent PES Solution) class diagram representation

The *PES Deployable Solution* is a Java class that encompasses all the information and specifications created along the PES Development by the PES designers. This information is kept stored during the PES Development, until the command to deploy the PES (which will be seen further ahead on the *Service Composition Engineering Tool*) is triggered, and a new instance of

this class is created and filled with it, and further sent to the Deployment environment. The *PES Deployable Solution* contains information such as the *PES Identifier (PES_Id)*, creator name, date of creation and the product/process associated to the PES. Also, it contains a list of *PESConfigurations*, which serve to incorporate the specifications for each selected resource intended to be used on deployment, as well for other resources with direct involvement on the deployment process (e.g. *Security Configuration* and the *Service Composition Configuration*). The *PES Deployable Solution* class also is provided with all the methods for handling the fields, either for inserting or extracting the values.

4.2.1.2. PES Configuration Class

The *PES Configuration* is a Java class that provides generic functionalities related to a given resource which is meant to be used on the PES Deployment (either if it's a *Deployable Service* or other component such as the *Service Broker* or the *Security Server*). For this, a Java class must be implemented and *extended* to the *PES Configuration* class.

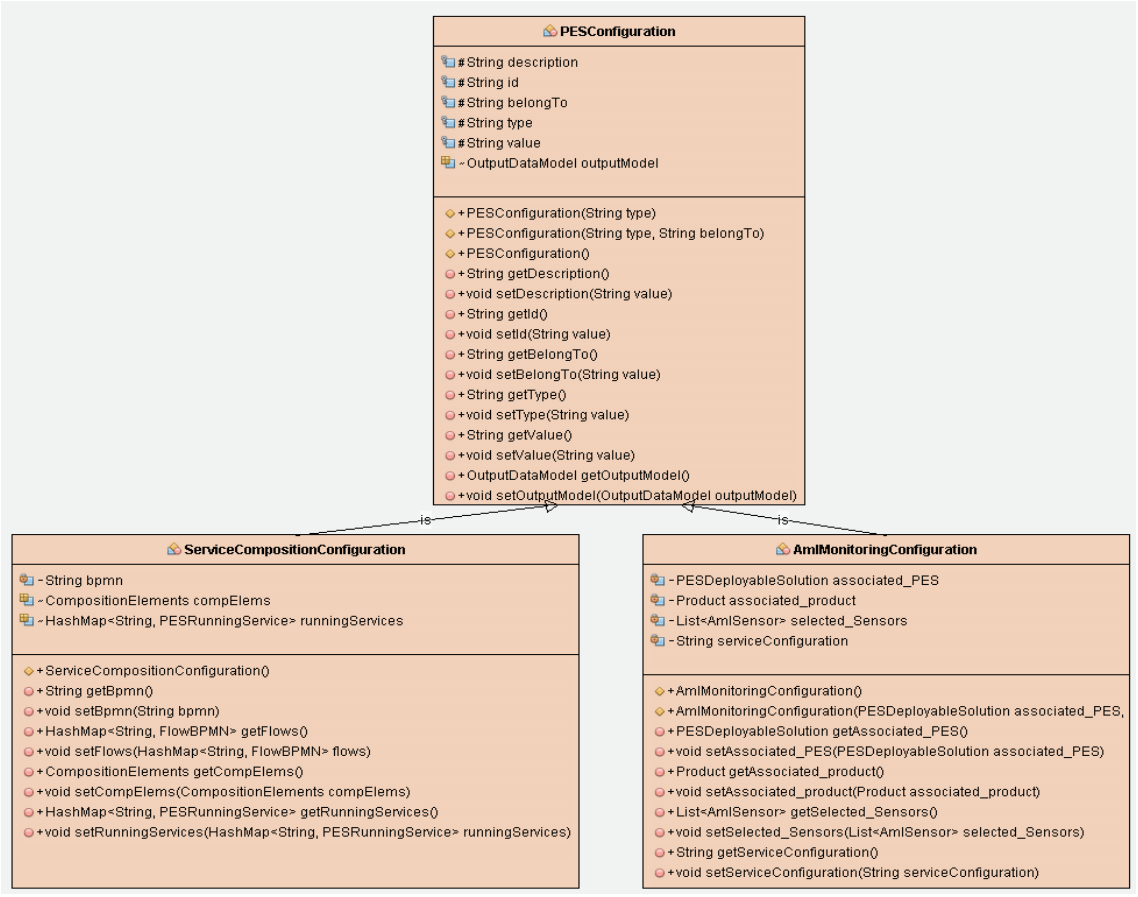

Figure 4.2 - PES Configuration class description, with two examples of sub classes (*AmIMonitoringComfiguration*, and *ServiceCompositionConfiguration*)

On Figure 4.2, it's visible two implementations of subclasses of the *PESConfiguration.* The *Service Composition Configuration* contains the *Service Composition* elements that will need to be passed to the *Service Broker* for launching the PES execution, and the *AmIMonitoring Configuration* includes the specifications for a selected *PES Deployable Service* to be used on the PES (in this case, a PES that will use the *AmIMonitoring Service*, will need to include the *AmIMonitoring Configuration* on the *PES Deployable Solution*, and later on the PES execution the respective service will need to receive it in order to be configured with the development specifications made for itself).

As seen in the *PES Deployable Solution* class, extending to a *PESConfiguration* enables that a set of configurations items are gathered into a list and passed from the development environment and the deployment environment. Still, in the case of the *PESConfiguration* being dedicated to a *PES Deployable Service* (as the *AmIMonitoringConfiguration* example), the generic fields present in the parent class are used by the system to identify and handle the object along the setup and execution of the PES. As example, the fields *config_Id, type*, *belongTo* and *OutputDataModel* are obligatory to be filled as they provide information to the deployment system of the, respectively, identification of the configuration, type of configuration, type of service to be allocated for the PES execution and the data model (as a Java class name) of the results produced by the service.

Also, due to constraints found on the self-mechanism of Apache CXF for binding messages (JAX-B) along the development, the field *JsonValue* is used to keep the serialized (in JSON format) configuration into a text field (*JsonString*), which was found the best way to pass around the problem.

4.2.2. Service Composition

ProSEco provides the mechanism for composing services that are encompassed in a certain PES, as demonstrated in chapter 3, involved in the PES Development process. For this, the *PES Development platform* offers an Engineering Tool – *Service Composition Tool* – which adds the environment for PES designers to compose, parameterize and deploy a PES. The *Service Composition Tool* is a web application, accessed at the end of the workflow of the PES development, which provides a Graphical User Interface (GUI) based on BPMN language. It has been selected a subset of BPMN (see Figure 4.3) that is capable of represent the service composition in a defined meta-language in use by the *ProSEco* system.

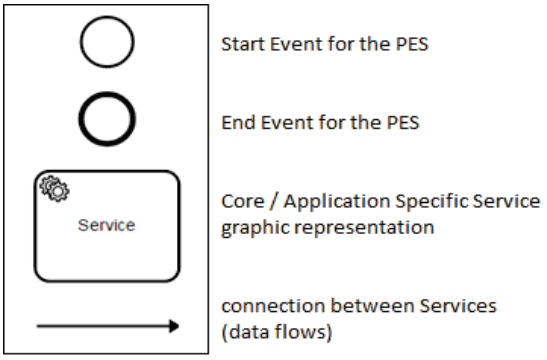

Figure 4.3 - Subset of the BPMN element used in ProSEco

The use of this elements provides most of the needs to satisfy the basic composition of the PES, however, to make the deployment of the PES able to be executed by the resources of the PES Deployment Platform, more information is needed to be added to the composition.

For this, another BPMN element is used – *Annotation* – which is used to encompass aggregated information for each of the selected Services.

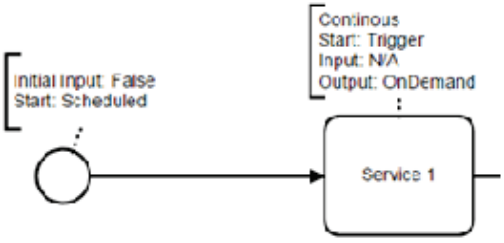

Figure 4.4 - BPMN Annotations element used on the service composition

4.2.2.1. Service Composition Process

At start up, the PES identifier (*Pes_id*) is obtained via the URL, or manually inserted, and used to query the *Knowledge and Management Base* (KMB) – a repository used by ProSEco system designed for store/get information associated to the PES Development – to retrieve the *PES Configurations* of the services in use for the related PES, and which were previously stored. The *PES Configurations* are retrieved and available in *Json* format, therefore the generic information can be analysed equally in all *PES Configurations*. In this case, the *PES Configuration* identifier (*config_id*) and the name of the service (type) are extracted and used to prepare the GUI on which the used can start the service composition process. As an example, considering that three services were selected for a PES – AmI Monitoring, Context Extraction and Data Mining –, meaning that there should exist three *PES Configurations* stored on the KMB repository at the time the tool is launched. The respective *Pes_Id* is used for querying the KMB repository and the *PES Configurations* are received, and analysed for constructing the service blocks of the GUI, while

passaging the respective identifier to the backlayer of the tool. As outcome, the GUI presented after the launching is showed in Figure 4.5:

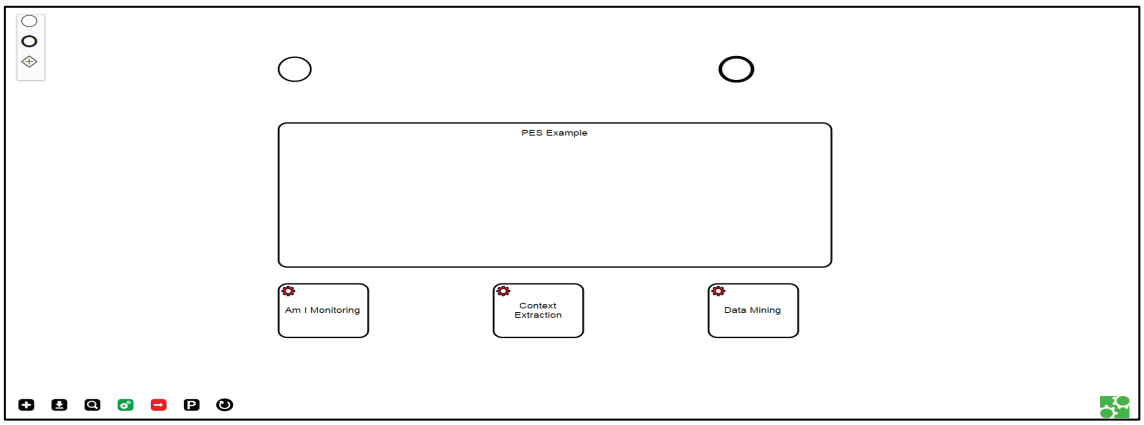

Figure 4.5 - Service Composition Engineering Tool User Interface on Start

The first step for the composing the services is to specify the data flows, by connecting the services boxes, defining for each flow who is the provider and the consumer of produced data while the PES is executing. An example of a well-formed composition is showed on Figure 4.6, where is visible that a Starting and Ending Service is defined, and there's no elements out of the PES area. Also, it's visible that each service can have more than one inputs or outputs for the dataflows, but that there are no duplicated flows.

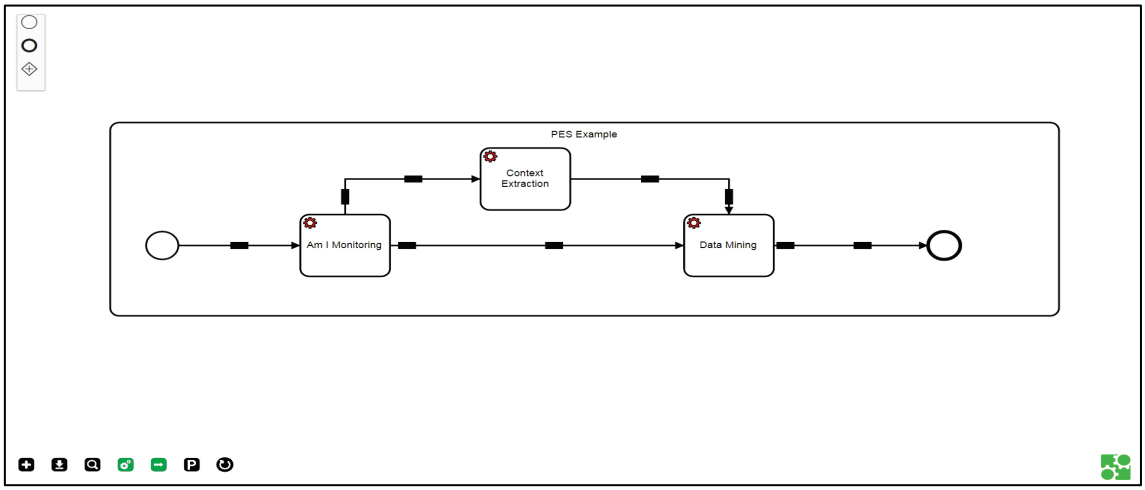

Figure 4.6 - Service Composition after the data flows specification

The second step of the service composition is to parameterize the data flows. Considering the *ProSEco* deployment environment, regarding each defined data flow, it's the consumer service that invokes (by calling a web-method for this purpose) the provider service. According to this, it has been induced that for each service that is a consumer, and for each data flow, it's necessary to specify:

- The time of the first invoke after the PES starts the execution
- The type of invocation:
    - *Periodic:* Where the system is aware and commands at given time, the service to pull data, regarding the respective flow (*Orchestration*)
    - *OnDemand:* Where it's the own service that will make the calls at his own specification, that is, the service must be implemented to be able to call for data without the intervention of the system (Choreography)

To accomplish the time parameterization of the data flows, the user must access, for each service, the respective form (see Figure 4.7) and fill the necessary fields according to the objective. When submitting each form, the information is settled on the BPMN annotation associated to the service (see Figure 4.8) on a specified format capable of being interpreted by the tool. All data flows need to be parameterized in order to conclude the service composition of the PES.

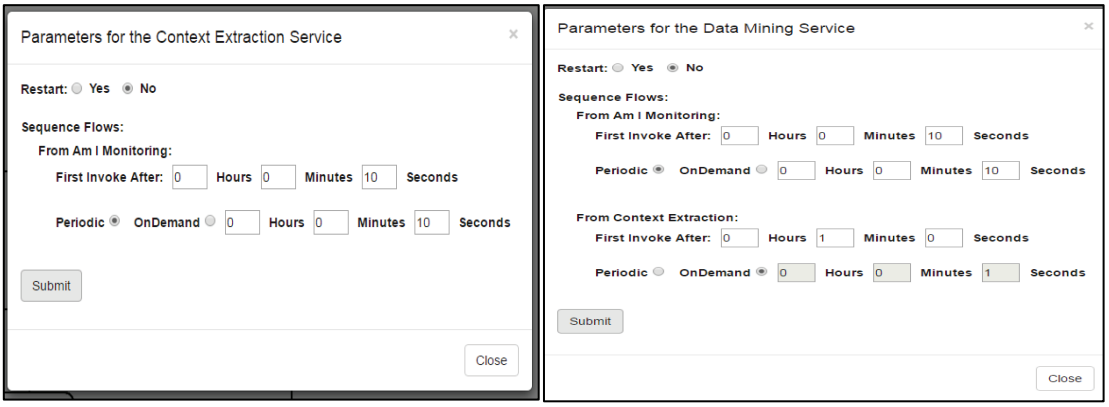

Figure 4.7 - Data flow parameterization pop-up form

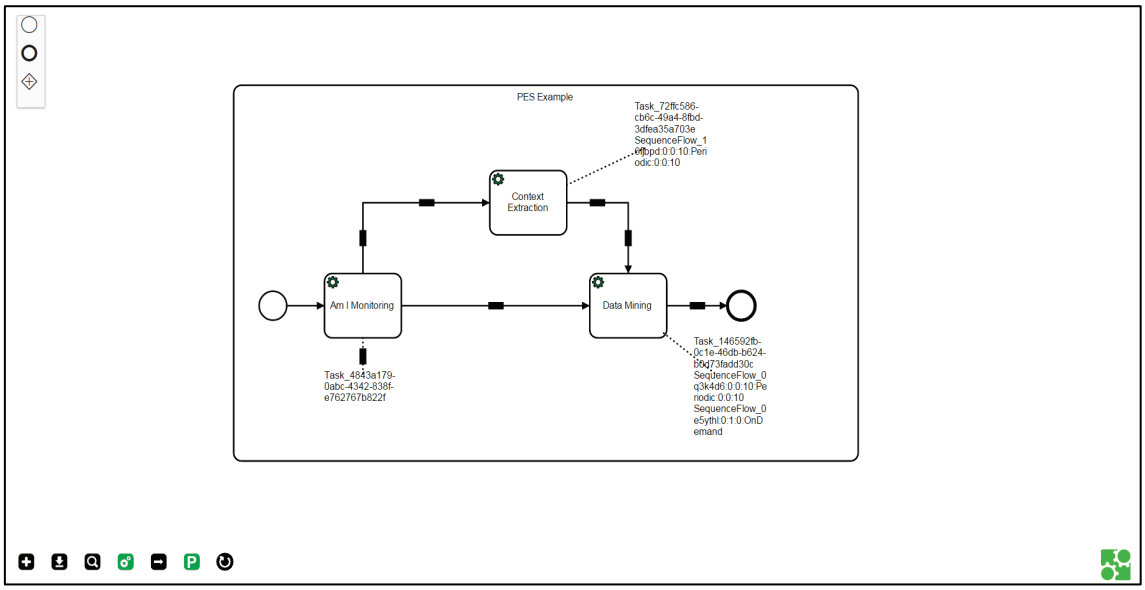

Figure 4.8 - Service Composition after data flow parameterization, leveraged by BPMN annotations

All the information created in the of the service composition process is encompassed in the Engineering Tool background as BPMN code in XML format, as seen in Figure 4.9, and can be retrieved and used at the moment the user wants to deploy the PES, activating the mechanism that the tool offers to create the *PES Deployable Solution* and send it to the *PES Deployment Platform*.

Figure 4.9 - Subset of the Service Composition BPMN code, in XML format

4.2.2.2. PES Deployment from the development environment

When the user wishes to deploy the PES, the *Service Composition* Engineering Tool will create and prepare the *PES Deployable Solution* so that it comprises all the PES information necessary for it to be deployed. At first, the tool will create a *ServiceCompositionConfiguration* (a *PESConfiguration*), which is intended to be interpreted by the Deployment environment.

4.2.2.2.1. Service Composition Configuration

This configuration aims to contain the service composition information that was developed, in a format that is easily interpreted and made use of, by the deployment environment. This is a Java class that is constructed based on the BPMN code produced by the *Service Composition* Engineering Tool which representation is showed on Figure 4.10:

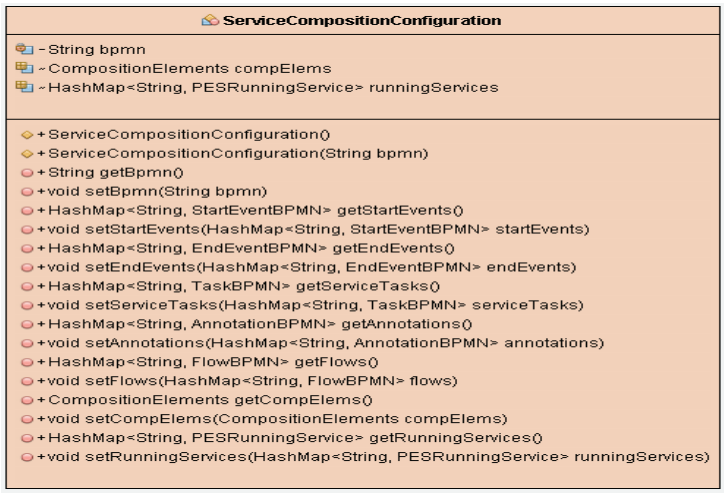

Figure 4.10 - *ServiceCompositionConfiguration* class representation

The *ServiceCompositionConfiguration* contains three fields:

- String *bpmn*: the BPMN code is inserted in the field, as a XML string, so that there's always a reference to the service composition original code
- Composition Elements *compElems*: Its a container for the BPMN elements, which are created while decodifing the BPMN code (see Figure 4.11). The BPMN elements are also Java classes specifically developed to individually translate each of the subset element of BPMN in use by the tool regarding the service composition that is being treated:
    - StartEventBPMN;
    - EndEventBPMN
    - TaskBPMN
    - AnnotationBPMN
    - FlowBPMN

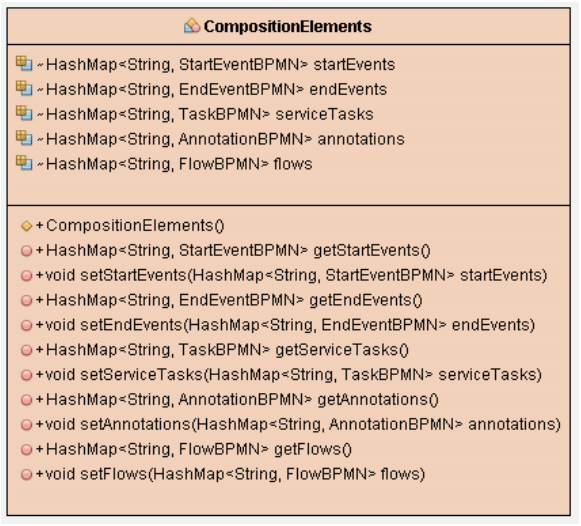

Figure 4.11 - *CompositionElements* class representation

- Map of *PESRunningServices*: This map contains the *PESRunningServices* objects, which are constructed based on the BPMN components (that is to say, on the service composition), regarding each of the selected services, meaning that for each service there must exist a *PESRunningService* object. The *PESRunningService* is also a Java class that contains all the necessary parameters that will be later be used on the *PES Deployment platform*, either by the agent-system or the resources (services) enabling the setup and the execution of the PES. For this, the *PESRunningService* is structured by fields that contain the service identifier (*ServiceID*) – which is the same of the respective configuration, the *PESConfiguration,* the data model (*outputDataModel*) defined for this service, the boolean value *isStarter* that informs if the service only as data provider in the scope of the service composition, and an HashMap of the *PESFlowSpecs*. By its turn, the *PESFlowSpecs*, are related to the data flows specified on the service composition presented on section 4.2.2.1, considering that each data flow for which the service stands as the receiver, have its parameterization contained in this class. In this sense, the *PESFlowSpecs* contain the following fields: the flow identifier *flowId*, the *flowtype* (that informs if the flow is *Periodic* or *OnDemand*), the time delay in regard to the PES execution start time *delayTImeOfPesStart* and to the data provider service *delayTImeFromSource* and the data model *OutputDataModel* in use for this specific data flow so that the service is capable of receiving the data in the correct format. Moreover, in case that the data flow is determined as *Periodic*, the period value and respective conversion into hours, minutes and seconds are included the numeric fields *period, hours, minutes* and *seconds*, respectively. On Figure 4.12, the class diagrams of *PESRunningService* and *PESFlowSpecs* classes are represented, with all the mentioned fields and methods that are used to manage and handle the class on its creation and usage on the PES execution.

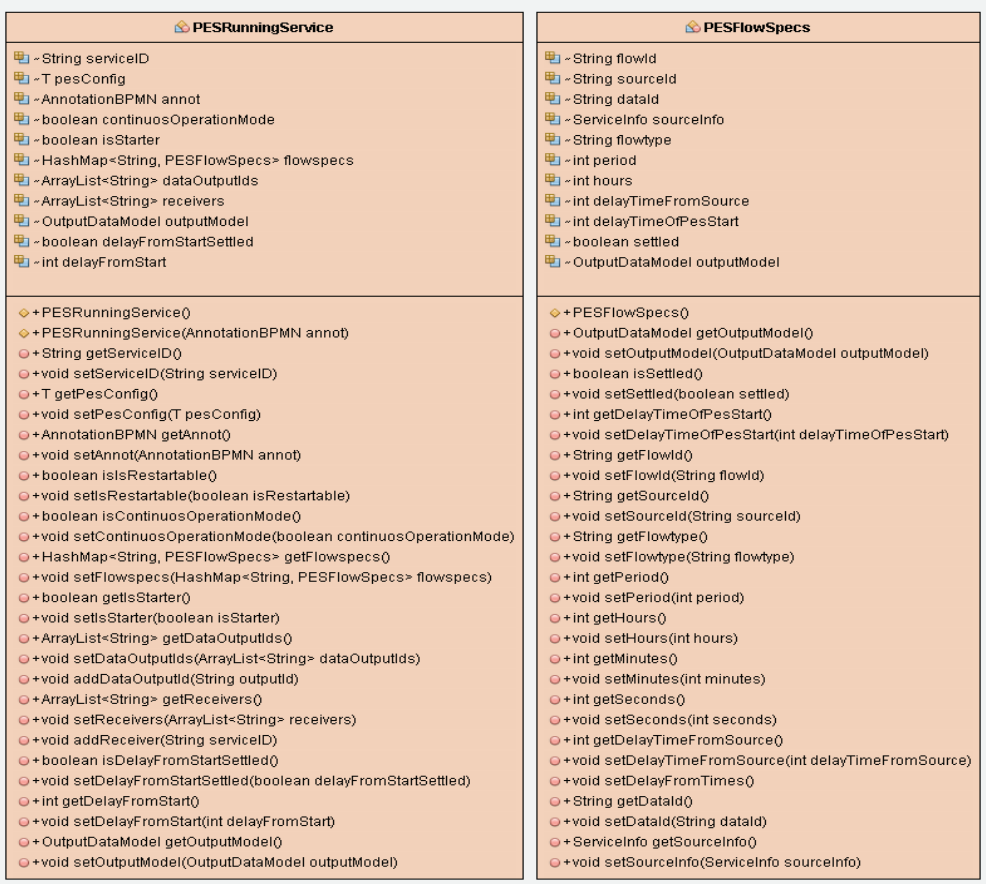

Figure 4.12 - *PESRunningService* and *PESFlowSpecs* classes representation

4.2.2.2.2. PES Deployable Solution Creation and Deployment

As already stated, when a PES is to be deployed, the *Service Composition* Engineering Tool is responsible for aggregating all the PES definitions created on the PES development into a *PES Deployable Solution* (see section 4.2.1.1). When the user activates the deployment, after the *ServiceCompositionConfiguration* is created, a new *PES Deployable Solution* is created with the respective Pes_id, name and date, and filled with all *PESConfigurations* (the previously retrieved and used on the service composition and, also with the recently created *ServiceCompositionConfiguration*).

Other fields from the *PES Deployable Solution*, such as creator name, are obtained by querying the KMB repository and added lately before deployment process begin.

Finally, the tool passes the *PES Deployable Solution* to the *Service Broker Service* (hosted on the PES Deployment platform), by invoking the *deploy* web method which the later provides. On Annex 1, it can be seen the *PES Deployable Solution* structure (translated in JSON format) that is created and sent to the *PES Deployment platform*, regarding the PES which was composed as

showed on Figure 4.13, and with the respective *PESConfigurations* inserted on the *hasConfigurations* field of the *PES Deployable Solution*.

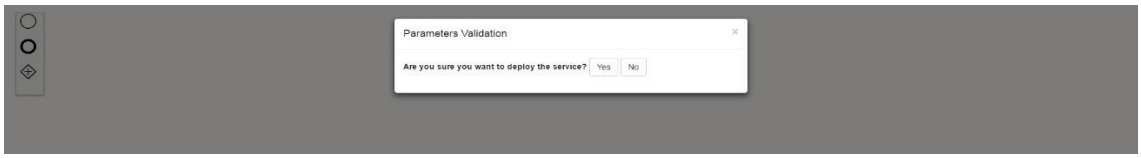

Figure 4.13 - *Service Composition Engineering Tool* PES deployment activation point

4.2.3. PES Deployment

The focus on this section is to detail the implementation of the *PES Deployment Platform* modules that enable the PES Execution process from the moment a *PES Deployable Solution* is received (as seen in the previous section) until the end of the execution, while following the structure and methodologies presented in Chapter 3. The Service Broker module (including the agent-based system dedicated to the PES deployment) and the generic adaptations made upon the resources (*PES Deployable Services* and *ProSeco Repositories*) are depicted along this section.

ProSEco provides a SOA infrastructure where the several modules are integrated as services, where the latest constitute the basic communication mechanism between the respective modules. Therefore, either in the PES deployment related or the own system related tasks, all the generated data during ProSEco platform lifecycle is passed from one component to the other by invoking the available services. Considering this, all the services implemented in *ProSEco* must meet two different requirements, namely:

- **Interface**: a Java interface must be specified, containing all the methods that the implementing service provides;
- **Implementation**: an actual implementation (Java class) must be provided, while inheriting and implementing the methods supplied by the interface

Moreover, for easing the integration of all the components of the architecture, the *IProsecoPrimitiveService* interface has been used as the top hierarchic interface, to whom every other service interface of *ProSEco* must implement. The *IProsecoPrimitiveService* interface provides the standard operations (see Listing 1) that can be used by all other services: *start*, *stop*, *restart* and *ping*. The *start* and *stop* methods are used to respectively start/stop the service, while *restart* is used to restart the service and usually in is only required when some configurations needs to be changed. The *ping* method is used to check the availability of the service to be reached.

Listing 1 – IProsecoPrimitiveService

```
1.  @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
2.  public interface IProsecoPrimitiveService {
3.
4.      @WebMethod(operationName = "startWebService")
5.      public void start() throws ProsecoFault;
6.
7.      @WebMethod(operationName = "stopWebService")
8.      public void stop() throws ProsecoFault;
9.
10.     @WebMethod(operationName = "restartWebService")
11.     public void restart() throws ProsecoFault;
12.
13.     @WebMethod(operationName = "pingWebService")
14.     public String ping() throws ProsecoFault;
15.
16. }
```

All components of the *PES Deployment Platform* are designed based on the service oriented principles leveraging the interoperability, re-usability and adaptability to different systems while enabling the integration of new types of services that can be added and used in the PES execution, standing as part of the infrastructure. As a result, all the components have to extend the *IProsecoPrimiveService* standard interface as the top-level interface, and implement the provided methods.

4.2.3.1. ProSEco Deployable Services

ProSEco Deployable Services are the distributed functional modules (web-services) available over the network that provide the core functionalities to the ProSEco system for execute the runtime of the designed and developed PES solution. The applied approach for developing the *ProSEco Deployable Services* is encompassed in a whole software solution – the Engineering Tool, the Service and the associated repository service.

The Engineering tool, as seen in section 3.3.1.1, is part of the *Meta Product & Process Development* platform, and used to create the respective *PESConfiguration*, for a given PES, that needs to be passed into the Service. On the other hand, the associated *Repository Service* is implemented to work together with the *Deployable Service*, as the latest will keep on producing results and storing them, following the methodology chosen by the Solution developer(s), in the associated Repository along the PES execution, making the results available to other services, as declared on the designed service composition of the PES. This fulfils the specific part of the implementation of the service, however, for enabling it to be handled inside the platform in the envisioned PES execution tasks, such as the setup of the service or to achieve the interoperability between the services, it was also found the need for services to be adopted of generic features that

leverage their integration as part of the workflow of the PES execution. For this, every *Deployable Services* needs to meet two several requirements:

- Implement the *IProSecoService* interface;
- Extend the *ProsecoDeployableService* abstract class

This way, the integration of a new resource in the platform is achieved by providing the implementation on the service, while adopting it of the inheritance structure visualized in Figure 4.14:

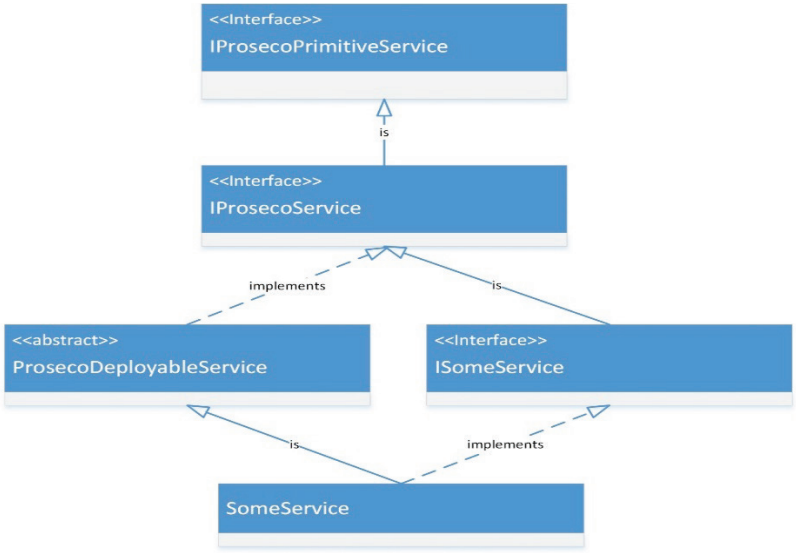

Figure 4.14 - Hierarchic inheritance of a ProSEco Deployable Service

The *IProsecoService* interface provides the necessary methods that the system must access at the given times while perform the PES setup and execution. But its relevant to consider that by only extending to the *IProsecoService* interface, the service developers would only have access to the placeholders of the inherited methods, being forced to implement them (or leaving them empty). Since, in accordance with the specifications of ProSEco, so that the resources are able to comply with the system and the service composition, an abstract class – *ProsecoDeployableService* – was implemented with the intent of adopt the services with generic fields and generic methods. Since the *ProsecoDeployableService* class itself implements the *IProsecoService* interface, then, by classifying (or not) the inherited methods with the *abstract* modifier, it's possible to make them as placeholders for implementation on the service class, or to provide them with a generic implementation that every class that extends to it will receive. On Listing 2, the specification of the *IProsecoService* interface is shown, while on Figure 4.15, the representation of the *ProsecoDeployableService* class is depicted:

Listing 2 – IProsecoService

```
1.  @WebService(name = "ProsecoService", targetNamespace = "http://proseco-
    project.eu/")
2.  @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
3.  public interface IProsecoService extends IProsecoPrimitiveService {
4.
5.      @WebMethod(operationName = "configureService")
6.      public <T extends PESConfiguration> boolean configureService(@WebParam(nam
    e = "Configuration") T Configuration) throws ProsecoFault;
7.
8.      @WebMethod(operationName = "getReposInfo")
9.      public ServiceInfo getReposInfo() throws ProsecoFault;
10.
11.     @WebMethod(operationName = "setupRuntimeSpecs")
12.     public boolean setupRuntimeSpecs(@WebParam(name = "host") String host, @We
    bParam(name = "port") int port, @WebParam(name = "classname") String className
    ,
13.             @WebParam(name = "dataOutputIds") ArrayList<String> dataOutputIds,
     @WebParam(name = "pesId") String pesId,
14.             @WebParam(name = "flowSpecs") HashMap<String, PESFlowSpecs> flowSp
    ecs, @WebParam(name = "serviceId") String serviceId, @WebParam(name = "outMode
    l") OutputDataModel outModel) throws ProsecoFault;
15.
16.     @WebMethod(operationName = "setNotifierClient")
17.     public boolean setNotifierClient(@WebParam(name = "host") String host,
18.             @WebParam(name = "port") int port, @WebParam(name = "classname") S
    tring className) throws ProsecoFault;
19.
20.     @WebMethod(operationName = "runtimeInvoke")
21.     public boolean runtimeInvoke(@WebParam(name = "flowId") String flowId) thr
    ows ProsecoFault;
22.
23. }
```

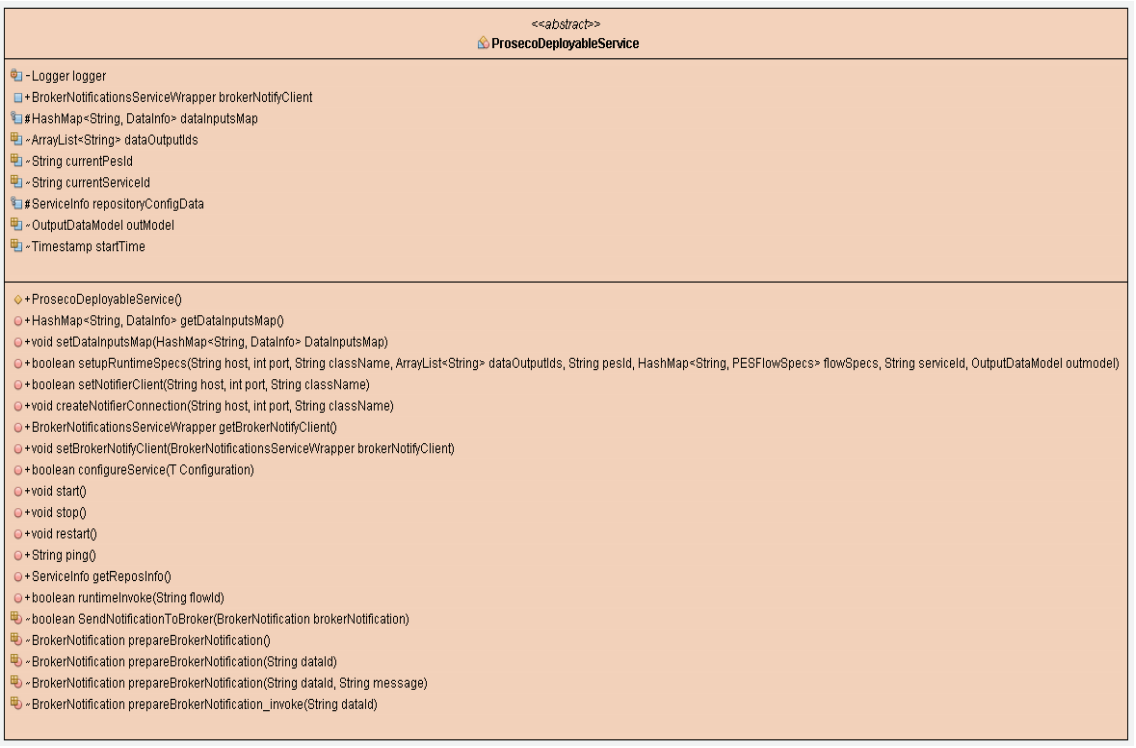

Figure 4.15 - *ProsecoDeployableService* abstract class representation

The *ProsecoDeployableService* provides its subclasses to inherit a set of generic fields that are used in respect to a given PES, considering that an instance of the service can only work exclusively for a PES at the time. This means that, at the time the PES execution starts, these fields need to be already filled accordingly. At first, following the methodology presented in section 3.3.3.1 – setup phase of the deployment -, the service is asked to pass the *RepositoryConfigData* field, which is a data structure that contains the location (as well as other specifications, such as the type of Service) of the associated repository, that needs to be passed into the system. Afterwards, the service is invoked to perform the setup, while it is passed all the information that will be casted into the fields: the associated *Pes_id* and *service_id* that are of the concern to the given PES, the *outputDataModel* where its specified the format of the results produced by the service, the *FlowSpecs* map that is translated to the *dataInputsMap* field, where the URLs of the *ProSEco Repositories*, flow identifier (*flow_Id*) and other specifications from where the service is intended to get data from are specified, and, the location of the *BrokerNotificationsService* in order to the service be able to send *Notifications* back to the agent-system. On Table 4.2 is explained how each of the inherited methods are used in regard to the PES Deployment.

Table 4.2 – Proseco Deployable Service inherited methods description

Proseco Deployable Service methods		
Web Method	**abstract**	**Feature**
start	y	It's invoked right at the time the PES enters the *Execution phase*, and is used for implementing any starting routine that the service needs
stop	y	Can be used same as start but for implementing stoppage routine (considering the previous example, it can be used to stop the active threads)
getReposInfo	n	When invoked, the service will reply with the data structure that contains all the necessary information of the associated Repository in use by himself
setupRuntimeSpecs	n	The service receives all the pre-treated information needed for executing the PES and uses it to fill and prepare the generic fields that will be used on the *PES Execution phase*
configureService	y	The service receives the respective *PESConfiguration*, and acts according to the implementation provided
runtimeInvoke	y	This call is received with an identifier of a data flow (*Flow_id*) at specified times, to let the service know that it's time to connect to the respective Repository, ask for data and act over the data received according to the implementation

Beside the methods inherited from the *IProsecoService* and *IProsecoPrimitiveService* interfaces, the *ProsecoDeployableService* class also supplies private methods to be used for convenience of the implemented service, such as the several *prepareBrokerNotification*, which are useful for creating standard *notifications* to be sent to the agent-system, or the *SendNotificationToBroker* which can be used at any time to send the *Notifications*.

4.2.3.1.1. Service Setup Process

According to the methodology presented in section 3.3.3.1.2, to prepare the *ProsecoDeployableService* for the PES Execution, it is necessary for it to receive the information that will allow it to accomplish the *Service Composition* related tasks. To do so, the *setupRuntimeSpecs* method is invoked, while passing the respective connectivity information that leverages the automatized connection and data requests from the other services.

The *dataInputsMap* Map will then be filled with *DataInfo* (see Figure 4.16) structures, one for each specified connection of the service composition, where the following fields are attributed:

- String *dataId*: the identifier of the specified flow (same as *flow_Id*)
- ServiceInfo *reposLocation*: information of the *ProSEco Repository* to be invoked, containing the URL and the type of repository
- String *flowType*: informing if the respective flow was designed as *Periodic* or *OnDemand*
- int *period*: the period in seconds. If *flowType* is designed as *OnDemand*, the value is 0.
- OutpuDataModel *outputModel*: contains the Java class that stands for the data model that will be received in this specific connection
- ProsecoRepositoryWrapper *service*: the actual client that connects to the ProSEco Repository Endpoint, which is assign based on the *reposLocation* field.

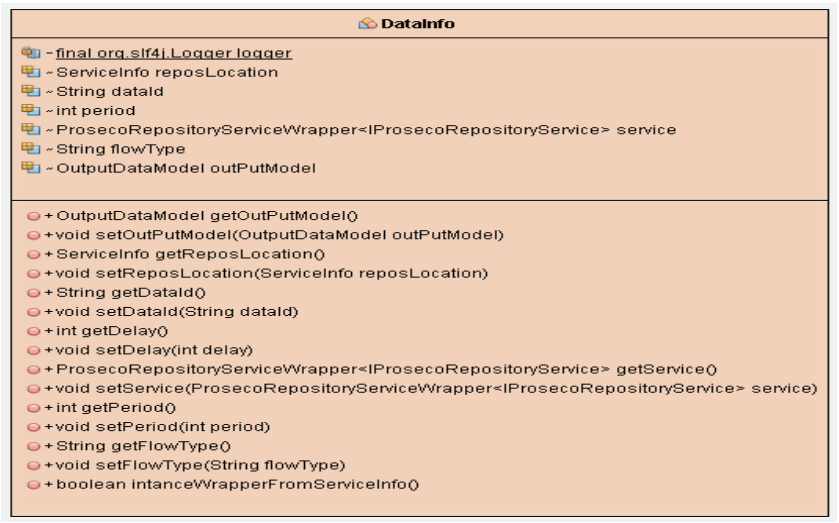

Figure 4.16 - *DataInfo* class representation

During the *Validation & Setup* phase of the PES, each Deployable Service will be reached through invocation of the *setupRuntimeSpecs* web method and perform the activity described in pseudo-code on Algorithm 1:

```
Require: BrokerNotificationService URL, Pes_identifier String, AssociatedService_identifier String, flowSpecsMap object
Ensure: boolean value
    Initialization;
        Assign Pes_identifier into Pes_id;
        Assign AssociatedService_identifier into service_id
        foreach element flowSpec of the FlowSpecsMap do
            create new DataInfo
            insert flowSpec elements into DataInfo;
            if repository of DataInfo can be ping then
                Insert DataInfo into DataInputsMap of DeployableService;
            else
                return false;
            end
            go to next flowSpec element;
        end
        if BrokerNotificationService endpoint can be pinged then
            Assign BrokerNotificationService into NotificationsWrapper
        else
            return false;
    return true;
```

Algorithm 1 – *setupRuntimeSpecs* method activity for setup a Deployable Service

To notice that, marked in red in Algorithm 1, for each defined data flow (*flowSpec*), the new *DataInfo* object is created, that uses the information received to try to establish the connection to the respective *ProSEco Repository*, by acting as presented in Algorithm 2, resulting on the service to acknowledge if the *Repository Service* is connectable:

```
Require: RepositoryInfo URL, flowId String, flowType String, OutputDataModel object, period int
Ensure: boolean value
    Initialization;
        Assign RepositoryInfo into repositoyLocation;
        Assign flowId into dataId
        Assign flowType into flowType
        Assign period into period
        Assign OutputDataModel into dataModel
        if RepositoryService endpoint can be pinged then
            Assign RepositoryService into RepositoryServiceWrapper
        else
            return false;
    return true;
```

Algorithm 2 - *DataInfo* creation and connecting to service on setup a Deployable Service

4.2.3.2. ProSEco Repository Services

The ProSEco PES Deployment Platform makes wide usage of repositories for storing fundamental data during the system lifecycle. In particular, the *ProsecoDeployableServices* must work with an associated repository that is a relational Data Base Management System implemented by using distinct technologies such as H2DB, MySQL, etc. The technology used to implement the repository is irrelevant since each database is wrapped into a service endpoint to allow the access, query and storing of the information. Therefore, a generic service that specifies all the web methods that the repository service provides, and which are directly involved in the PES deployment has been provided. The methodology is similar to the one of the *ProsecoDeployableServices*, that is, an interface – *IProsecoRepositoryService* interface - has been provided for declaring the methods, and an abstract class named *ProsecoRepositoryService* provides the generic fields and methods which all repositories classes must, respectively, implement and extend in order to inherit. As result, the integration of any ProSEco repository in the system follows the structure presented on Figure 4.17, while fulfilling the next requirements:

- Implement the *IProSecoRepositoryService* interface;
- Extend the *ProsecoRepositoryService* abstract class

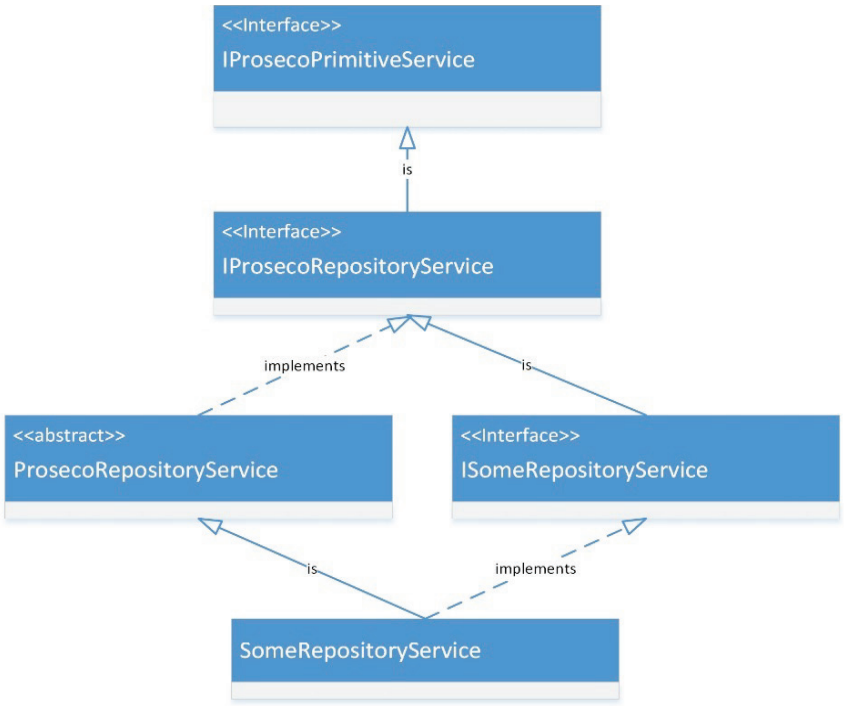

Figure 4.17 - Hierarchic inheritance of a ProSEco Repository Service

The *IProsecoRepositoryService* interface (see Listing 3) declares the methods to be accessed by other components of the system:

Listing 3 – IProsecoRepositoryService

```
1.  @WebService(name = "ProsecoRepositoryService", targetNamespace = "http://prosec
    o-project.eu/")
2.  @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
3.  public interface IProsecoRepositoryService/*<T extends FlowController>*/ extend
    s IProsecoPrimitiveService {
4.
5.      @WebMethod(operationName = "storeElement")
6.      public String store(@WebParam(name = "Element") Object Element) throws Pros
    ecoFault;
7.
8.      @WebMethod(operationName = "removeElement")
9.      public boolean remove(@WebParam(name = "Element") Object Element) throws Pr
    osecoFault;
10.
11.     @WebMethod(operationName = "getElementbyID")
12.     public String invokeForData(@WebParam(name = "ElementId") String ElementId)
     throws ProsecoFault;
13.
14.     @WebMethod(operationName = "setOutputModel")
15.     public void setOutputModel(@WebParam(name = "OutputModel") OutputDataModel
    model) throws ProsecoFault;
16.
17.     @WebMethod(operationName = "setOutputIds")
18.     public void setOutputIds(@WebParam(name = "OutputIds") ArrayList<String> ou
    tputIds) throws ProsecoFault;
19.
20.     @WebMethod(operationName = "setupRepos")
21.     public boolean setupRepos(@WebParam(name = "host") String host, @WebParam(n
    ame = "port") int port, @WebParam(name = "classname") String className, @WebPar
    am(name = "pesId") String pesId, @WebParam(name = "model") OutputDataModel mode
    l, @WebParam(name = "outIds") ArrayList<String> outIds, @WebParam(name = "servi
    ceId")String serviceId)   throws ProsecoFault;
22.
23.     @WebMethod(operationName = "startPES")
24.     public boolean startPES()  throws ProsecoFault;;
25.
26. }
```

The abstract class *ProsecoRepositoryService*, represented in Figure 4.18, provides the generic fields and necessary abstraction to the methods which need to be implemented while leaving the ones non-marked with *abstract* modifier with generic implementation. Concerning the PES Execution, the *ProsecoRepositoryService* class is provided of the following fields:

- *dataOutputIds*: a vector with the identifiers (*flow_Ids*) of the flows who are meant to ask for data, in accordance to the designed service composition
- *model*: the *OutputDataModel* structure that was defined by the respective Engineering Tool, to provide the recognition of the data by the other services who will receive the produced results

- *flowControls*: a Map of *FlowControllers,* containing one controller for each identifier defined on the *dataOutputIds* value
- *startTime*: the numeric start time (Timestamp)
- *currentPesId*: the identifier of the PES for which the repository is dedicated to
- *associatedServiceID*: the identifier of the associated *Deployable Service*
- *brokerNotifyClient*: The client for the Notifications Service endpoint, used for sending *Notifications* back to the agent system

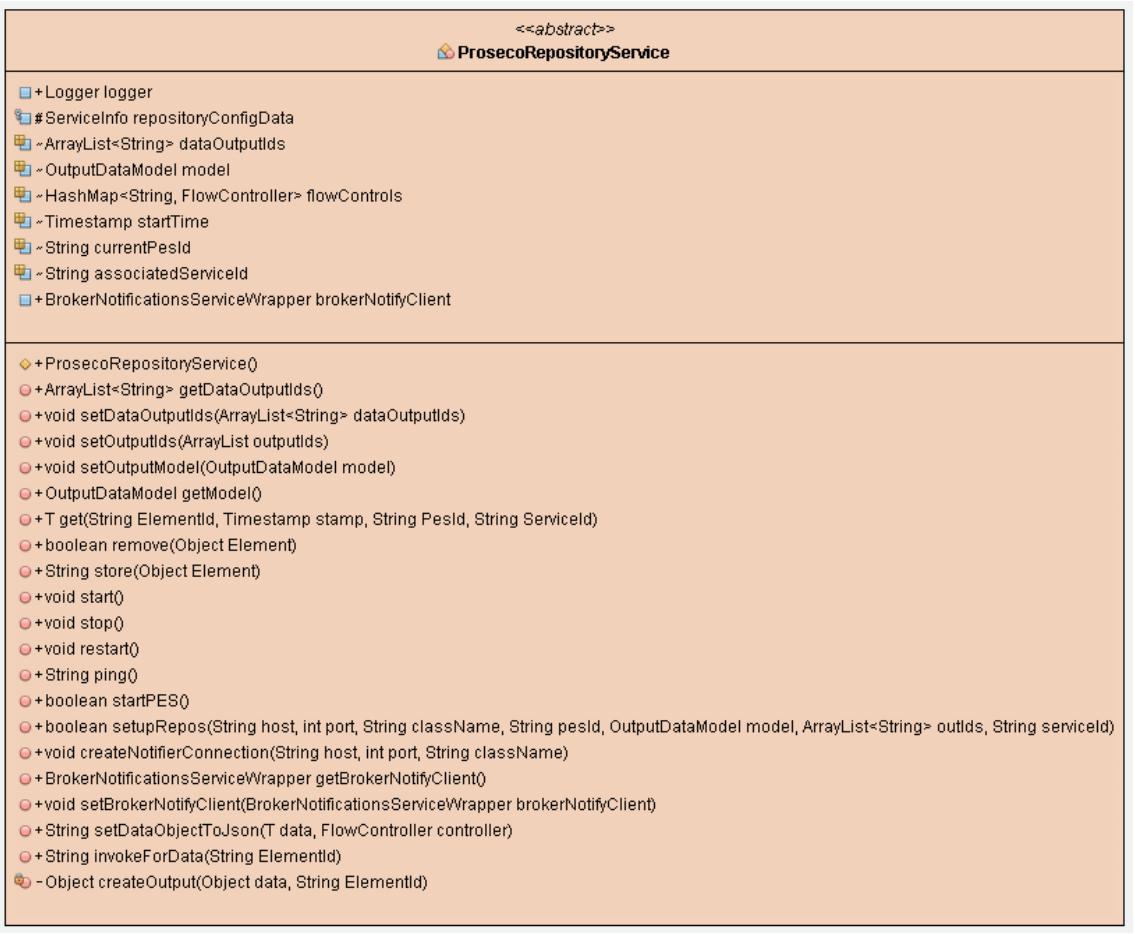

Figure 4.18 - *ProsecoRepositoryService* abstract class representation

Regarding the inherited methods implemented for repositories, Table 4.3 provides a description of the relevant ones concerning the PES Execution:

Table 4.3 – Proseco Repository Service inherited methods description

Proseco Deployable Service methods		
Web Method	**abstract**	**Feature**
Store	y	Used by the associated Service to store the results that are produced along the PES Execution. It's open for implementation, as the service is aware of the identifiers of the PES, Service, and Flow, and can choose the methodology to adopt on storage process
Remove	y	Can be invoked by the associated Service to remove any set of stored results. Also open for implementation for the service developers to adopt their own methodology
setupRepos	n	The repository service receives all the pre-treated information needed for executing the PES and uses it to fill and prepare the generic fields that will be used on the *PES Execution phase*
startPES	n	Informs the repository service that the PES Execution has begun.
InvokeForData	n	The method that is triggered by a Deployable Service, in order to gather and send the results, according to the *flow_id* received, while using and updating the respective *flowController*.
Get	Y	This method is open for implementation in order to extract the results, as it receives the PES, service and flow identifiers and also the date of the last extraction. Every parameters may be used in the applied methodology for gathering the data.

4.2.3.2.1. Repository Setup Process

Once again, the implementation done on the *ProSEco Repositories* is very similar to the one of the *ProSeco Deployable Services*, while gearing into the aim of the component, and following the methodology presented in section 3.3.3.1.2. The main *ProSEcoRepositoryService* method invoked for triggered the setup dedicated to a PES is the *setupRepos* web method. During the *PES Validation & Setup* phase every repository involved in the PES is reached through this method and receives the following parameters:

- String *pes_id*: the identifier of the PES
- String service_Id: the identifier of the associated service
- OutputDataModel *model*: the data model for the results to be provided and sent to the other services

- Array of Strings *outIds*: the data flow identifiers (*flow_ids*) expected to be during the PES Execution, according to the designed service composition
- URL *BrokerNotificationServiceURL* (separated in host, port and name): to construct and connect to the *Broker Notifications Service*.

The *pes_id*, *service_id* and *dataModel* are assigned to the respective existing fields of the *ProsecoRepositoryService* class. Afterwards, for each *flow_id* that comes on the *outIds* array a new *flowController* is created and inserted into the *flowControls* map, for later to be used on the *PES Execution* phase and which will be seen in the further sections. At last, also the *Repository Services* must be able to connect to the agent system by instantiating a client for the *BrokerNotificationsService* endpoint, so the parameters for mounting the URL are also sent by the system. The method will try to *ping* the *BrokerNotificationsService* and replies by sending the Boolean value *true* or *false* according to the success of the operation. This method is described in pseudo-code in Algorithm 3:

```
Require: BrokerNotificationService URL, Pes_identifier String, AssociatedService_identifier
String, outIds Array of String, dataModel OutputDataModel object
Ensure: boolean value
        Initialization;
                Assign Pes_identifier into Pes_id;
                Assign AssociatedService_identifier into service_id
                Assign dataModel into outMode
                foreach element flowId of the flowIds Array do
                        create new FlowController
                        insert model elements into flowController;
                        Insert FlowController into flowControlsMap;
                        go to next flowId element;
                end
                if BrokerNotificationService endpoint can be pinged then
                        Assign BrokerNotificationService into NotificationsWrapper
                else
                        return false;
        return true;
```

Algorithm 3 – *setupRepos* method activity for setup a Repository Service

Once the *setupRepos* method in concluded, the repository will keep in standby until the *startPES* method is invoked, indicating that the *PES Execution* has been ordered to start. Considering this, the *startPES* activity, seen in Algorithm 4, consists on acknowledging the current time and assigning it to the *startTime* field of the *ProsecoRepositoryService* class. Finally, the obtained *startTime* is assigned to all the *flowControllers* previously created on the setup.

Require: *setupRepos* method concluded
Ensure: boolean value
Initialization;
Get *current time* as Timestamp
Assign *current time* **into** *startTime*;
foreach *flowController* of the *flowcontrols Map* **do**
insert *startTime* **into** *lastReadTime*;
go to next *flowId* element;
end
return *true*;

Algorithm 4 – *startPES* method activity for setup a Repository Service

At this point, each of the *flowController* is then prepared and ready to be used along the PES Execution. The *flowController* class is composed by the respective flow identifier (*flow_id*), *OutputDataModel* structure and the date of the last request (Timestamp *laststep*). These fields and the respective handling methods of the *flowController* class are used at the PES Execution to make the time control of the data request, as will be seen in the next section, which concerns to the interoperability between services.

On Figure 4.19, the *flowController* class is represented:

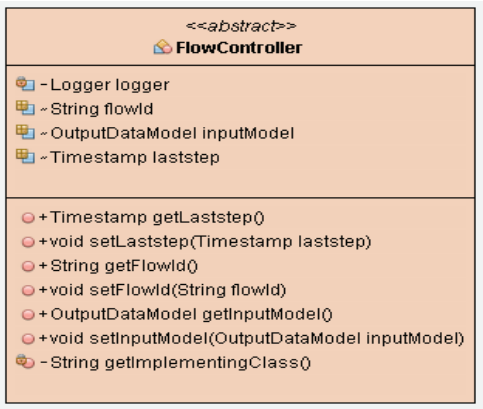

Figure 4.19 - *flowController* class respresentation

4.2.3.3. Interoperability of Services on PES Execution

The usage of the SOA paradigm inside ProSEco is not only applicable to the ProSEco System structure, but also to the PES workflow oriented to the fulfilment of the proposed objectives. SOA allied with the Service Composition served as the background to development and implementation of the methodology presented in Chapter 3, which leverages the communication

and data transactions between the resources involved in a PES Execution. In this sense, it's right to affirm that the interoperability between the resources is a main feature that accomplishes the PES objectives, keeping untouched the individuality of those resources but letting them collaborate towards the PES goals. To demonstrate the implementation made to fulfil the generic methodology for the interoperability of services, consider the Service Composition seen in Figure 4.20:

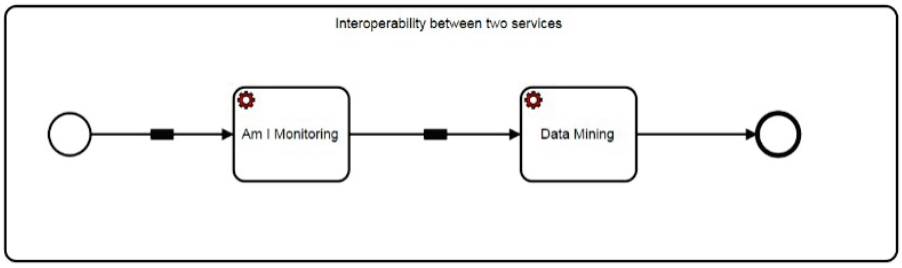

Figure 4.20 - Service Composition of a PES with two services

In this simple example, regarding the designed data flow, the *AmIMonitoring* service stands as the *Sender*, while the *DataMining* service stands as the *Receiver*. From the point of view of the *PES Deployment platform* components, this is equivalent to what is seen in Figure 4.21:

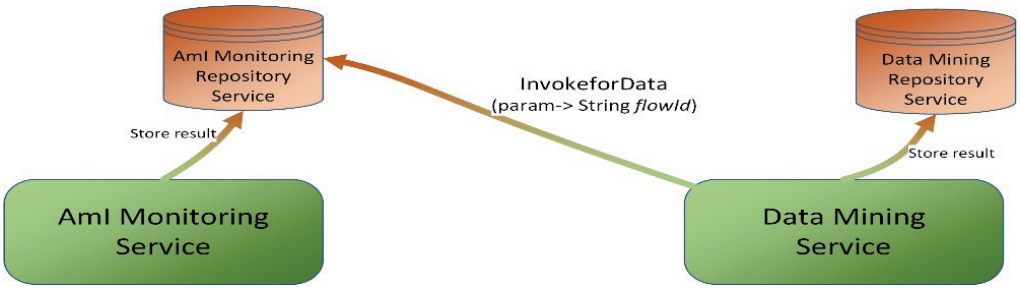

Figure 4.21 - Applied communication mechanism between two services

It's visible that the *InvokeforData* method of the *ProsecoRepositoryService* is the trigger to commence the data transaction process. The *Receiver* service, in accordance to the specification of the *Service Composition*, either by order of the agent system or self-demanded, calls the method while sending the identifier of the data flow (*flow_Id*), at any time, once the *PES Execution* has started. As seen in the sections 4.2.3.1.1 and4.2.3.2.1, all *Deployable Services* and *Repository Services* are already setup at this phase, meaning they already possess all the information that allows the completion of the task.

4.2.3.3.1. Sender Service Interoperability Process

The Sender Service (*Repository Service*) begins the workflow for gathering the data by accessing the respective *flowControler* from the *flowControlsMap*, by using the received *flow_Id*. In case there's no mapping into any value of the Map means that the identifier was not specified for any data flow, and therefore the task is ended. If a match is found, the *lastStep* value containing the time of the last call is acknowledged and the *Get* method is started, remembering that this method is a placeholder for the developer(s) to implement, and where the following arguments are passed: *lastStep*, *Pes_Id*, *Service_Id* and *flow_Id*. The *Get* method must be implemented to return a Java object according to the specified *OutputDataModel* in order to perform the automatic serialization of the result object. Finaly, the *flowController* is updated with the current date (defined at the beginning of the method) before sending the serialized result back to the *Receiver*. This workflow is demonstrated in pseudo-code on Algorithm 5:

```
Require: flow_Id String
Ensure: finalResult Serialized result object
    Initialization;
        Get flowController from controlsMap equals flow_Id ;
        if no flowController found then
            return null; // Exit method
        end
        Create currentTime date object;
        Assign lastTime date object from flowController;    // lastStep field
        Call get() mehtod with arguments: flow_Id, Pes_id, service_Id, lastTime;
        Assign return value from get() method into rawResult;
        Assign finalResult from rawResult Serialization process;
        Assign currentTime into lastStep from flowController; // controller update
    return finalResult;
```

Algorithm 5 – *invokeForData* method activity of Repository Service

On Algorithm 5, the task for Serializing the result obtained by the *get* method is marked in red, due to a restraint of the system found while developing the ProSEco prototype. The services implementation relies on APACHE CXF, which uses JAX-B internal library for binding the data exchanged on the methods invoked. It was discovered that the XML based serialization done by JAX-B was not capable of preventing some faults that occurred while assigning the objects passed on the web method call, when the object was from (or contained other objects that were from) certain class types, as for example, the *TimeStamp* Java class. After studying and considering the options to work around this issue, and considering that the system needs to adapt to any kind of objects that are exchanged without previous knowledge of it, it was found that the most efficient

and easy option was to make a pre-Serialization of the data using JSON technology. The following advantages were found by using this option:

- Easy implementation for Serialization and de-Serialization
- The object passed on the reply to the invoked method is always a String, which is considerably less susceptible to errors
- Even not knowing the Java Class of the results, the object is assigned into a JSON Object with the ability to be introspected

As so, the methodology presented on Algorithm 6 was implemented, where the *rawResult,* which is of Java Object whose type is known through introspection of the *OutputDataModel*, is converted into a JSON String, for after being sent as the final result.

Require: *rawResult* Java Object, *dataModel* OutputDataModel object
Ensure: *finalResult* JSON object
    Initialization;
        Get *className* **from** *dataModel*;
        if *className exist* **then**
            Create *result* as JSON String **from** *rawResult* and *className*;
            Assign *Result* as JSON String **into** *finalResult*;
        else
            Create *result* as JSON Object
            Assign *Result* as JSON Object **into** *finalResult*;
        end
    return *finalResult*;

Algorithm 6 – *SetResultIntoJSON* method activity

4.2.3.3.2. Receiver Service Interoperability Process

In order to leave every service open for implementation according to their specific aims, the *runtimeInvoke* method has been marked as an *abstract* method, meaning each developer needs to implement it towards their desire. However, it has been implemented a generic case, applicable to every service, that allows receiving and collecting the results from a given data flow into their original format (specified in the respective *OutputDataModel*). For this, every time the *ProsecoDeployableService* decides or is commanded to pull data from other service, it will use the respective flow identifier (*flow_Id*) which was assigned on the Service Composition. If the service has been correctly with the data flow specifications, the *flow_Id* is used to consult the *dataInputsMap* and retrieve the *DataInfo* object (as seen in section 4.2.3.1) so that all the information retained about the respective data flow can be used. From here, the service creates a connection to the *ProsecoRepository* using the URL from the *DataInfo* object and invokes the

invokeForData method, seen in the previous section, passing the *Flow_Id* as parameter. Then, in case the result is not empty (*null*), remembering that the result should be provided as an JSON object, the *OutputDataModel* from the *DataInfo* object is analysed. If a data model class has been specified, then service tries to de-serialize the result into an object of the data model type. In case a fail occurs, or if the data model is not specified, the service will create an *JSON Object* from the result. At this point, the implementation of the service may be defined to use the result. On Algorithm 7, the described workflow is represented in pseudo-code:

```
Require: flow_Id String, command to start the method
Ensure: result Object
    Initialization;
        Get  DataInfo object  from dataInputsMap of service where flow_Id match;
        if DataInfo exist then
            Create RepositoryService client from URL of DataInfo;
            Call invokeForData of client with flow_Id parameter;
            Assign invokeResult reply to JsonResult;
        else
            return false; (Exit method)
        end

        Get  dataModel class name from DataInfo object;
        if dataModel exists then
            Set result as dataModel object from JsonResult and dataModel class
            name;
            if result not succesfully set then
                Set Result from JsonResult as JSON Object;
            end
        else
            Set Result from JsonResult as JSON Object;
        end

        ##################
        #
        #  Service implementation for using the received data…
        #
        ##################

    return true;
```

Algorithm 7 – *runtimeInvoke* activity for preparing the result from other service for usage

To have a better overview how the interoperability is done, on Figure 4.22 it's represented the workflow of two *PES Deployable Services* and associated *Repository Services* allocated to a given PES, that were composed to perform data exchange along the PES execution. In this case, Service A stands for the *Sender Service*, as Service B represents the *Receiver Service*, which on the examples provided in Figure 4.20 and Figure 4.21 are, respectively, the *AmI Monitoring* service and the *Data Mining* service:

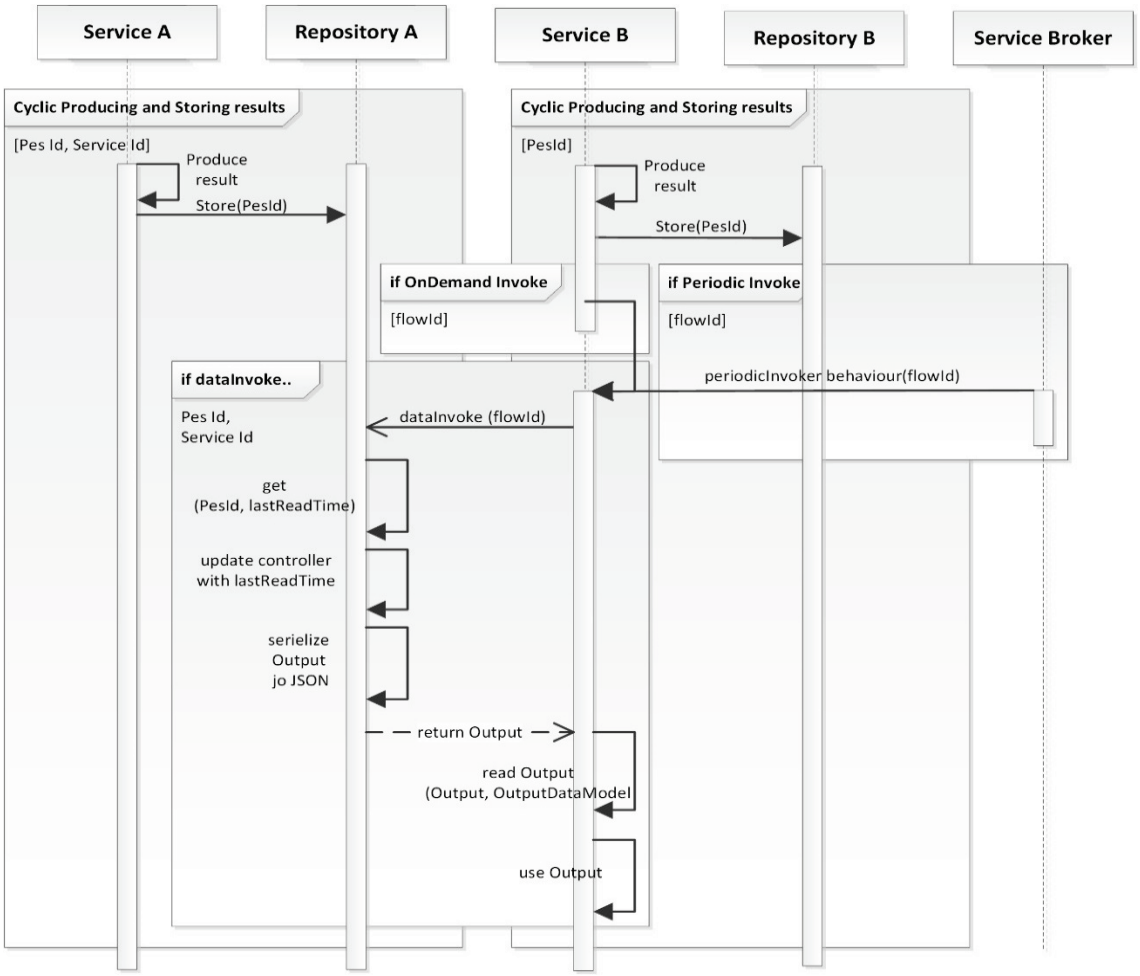

Figure 4.22 - Data exchange between services along a PES Execution

4.2.4. Agent System implementation

As seen in Chapter 3, the Service Broker component was developed in order to provide the main features of the system that leverage the PES Setup and Execution control, being that is structured by two main components:

- *Service Broker Service*: the endpoint of the system for outer communication, enabling the receival of new PESs and requests for data that is to be presented in the *Service Broker* GUI
- *Agent system*: An internal system that is responsible for performing all the logic inherent to the deployment of PESs that were *developed* and sent to the *PES Deployment platform*

The focus of this section is to deliver the implementation of the *Service Broker Agent System*, which follows the architectural and methodological specifications presented in Sections 3.3.2.5

and 3.3.3, and which enables the system to act upon the receival of new PESs, and throughout the PES deployment stages, namely:

- *Validation & Setup phase*: where the received *PES Deployable Solutions* are interpreted and handled to by the agent system in order to validate the PES and setup the resources to be used
- *PES Execution phase*: where the agent system monitors the resources that perform the PES Execution by means of dedicated agents which reproduce the specific functionalities of each of the actors involved on the PES deployment

4.2.4.1. Agent System Framework

The development of the Agent-based system relies on Java Agent Development Framework (JADE), once the proposed architecture of ProSEco being built over JAVA language, leading to an easy integration process. Also, JADE offers a flexible domain-independent infrastructure that leverages the development of complete agent-based applications and systems, and is provided of several auxiliary tools and internal libraries that facilitates the implementation of new agents, associated logic and communication.

4.2.4.1.1. Agent Communication

As agents are autonomous entities inside the environment where they habit, the inter-agent communication is placed as a main feature that allows each agent to be aware of the surrounding progress, so that the decision making and consequent action can be taken.

Several FIPA protocols are available for use with JADE, as the implemented agents are FIPA compliant. For the current specifications of ProSEco, the *FIPA Request Protocol* was found to serve the needs implied by the system as this protocol consists on point-to-point communications, where an agent is able to reach out any other agent through an ontology based message system, letting the requested agents to forward into the necessary response to the received content.

The *FIPA Request Protocol* is initiated by the *Initiator* agent, which prepares and sends a request message to the participant(s) agent(s). Each participant may accept or refuse by sending a first reply to the *Initiator* (*agree* and *refuse*). In case the participant agrees, it perform the necessary action and sends a second reply where it identifies the result of the operation. In case the action was successful, the reply must be of *inform* type. Otherwise is shall send a *failure* note back to the *Initiator*.

In Figure 4.23, the above FIPA Request Protocol is described:

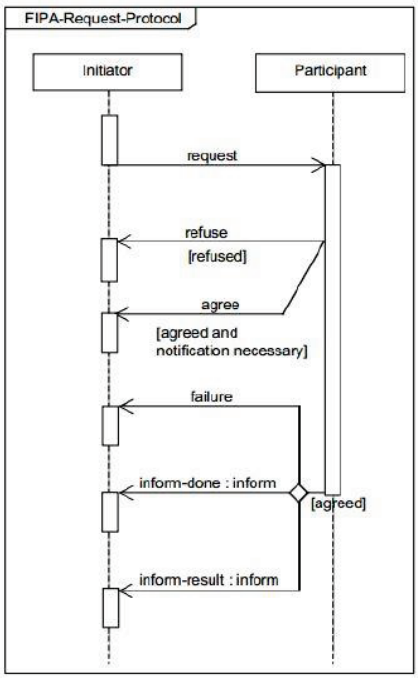

Figure 4.23 - FIPA Request Protocol

4.2.4.2. PES Validation & Setup

As seen in Section 3.3.2.5.3, three types of agents with specific orientation of work are considered for the system, where they co-habit independently but somehow represent the hierarchy of the actors involved on the PES deployment:

- *Broker Handler agent*: as top-level agent, represents the *PES Deployment platform*, as it's oriented to handle the incoming requests or notifications of the PESs (e.g. when a new PES is received or when a notification from a resource that is executing a PES is received);
- *PES Processor agent*: middle-level agent that is created for each received PES, and that is responsible for handling the respective *PES Deployable Solution* by analysing it and forwarding the system in the necessary direction;
- *Runtime Service agent*: the low-level agent that stands for each of the resources (*PES Deployable Service*) in use for the PES Execution, seen by the system eyes.

Agents are recognized by acting through implemented behaviours who are triggered by some event and usually perform a decision over the input received. From the point of view of the *PES Deployment*, the first agent to act is the *Broker Handler agent*, as it receives the request to start a new PES, incoming from the Service Broker endpoint.

4.2.4.2.1. Broker Handler Agent

The *Broker Handler agent* class, represented in Figure 4.24, contains information of the system that is required in the *PES Deployment* process by other agents, such as the *Service Registry Service* and the *Broker Notifications Service* endpoints location. Also, this agent is responsible for launching the *PES Process agents* while keeping their reference in a *HashMap* (*processAgentMap*), for future communication.

Figure 4.24 - Broker Handler agent class representation

The main behaviours of the *Broker Handler agent* regarding the *PES Deployment* are dedicated to start a new PES (*startNewPES* behaviour), launch the *PES Processor agent* (*ProcessAgentStartInitiator* behaviour) and to stop the PES (*DeleteProcessAgent* initiator). These behaviours can be depicted in Figure 4.25.

The *startNewPES* behaviour is activated when a *PES Deployable Solution* is received. The agent will extract the PES identifier (*Pes_Id*) from the *PES Deployable Solution* and consults the *processAgentMap* for a match, as the launched *PES Processor* agents are registered with the correspondent *Pes_Id*. A match found means that the PES is already in deployment, so the agent will prevent from continuing the process. Otherwise, a new *PES Processor* agent is launched and passed the *PES Solution* into, while inserting the new agent in the *processAgentMap* and activating the *ProcessAgentStartInitiator* behaviour.

The *ProcessAgentStartInitiator* behaviour follows the *FIPA Request Protocol*, and as so, once the request is sent to the respective *PES Processor* agent, it will keep on hold until receive a reply. The reply type enables the *Broker Handler agent* to acknowledge if the *PES Processor* was successfully launched and setup. In case the reply is an *inform*, the operation was successful. Otherwise, if the reply is from *failure* type, means the *PES Processor* agent was not able to perform its task, so the *DeleteProcessAgent* initiator is triggered.

The *DeleteProcessAgent* initiator method sends a request to the *PES Processor* agent, triggering its termination routine and deletion from the system. Once the task is completed, an *inform* is received on the reply and the *processAgentMap* is updated (removal of the reference of the respective *PES Processor* agent).

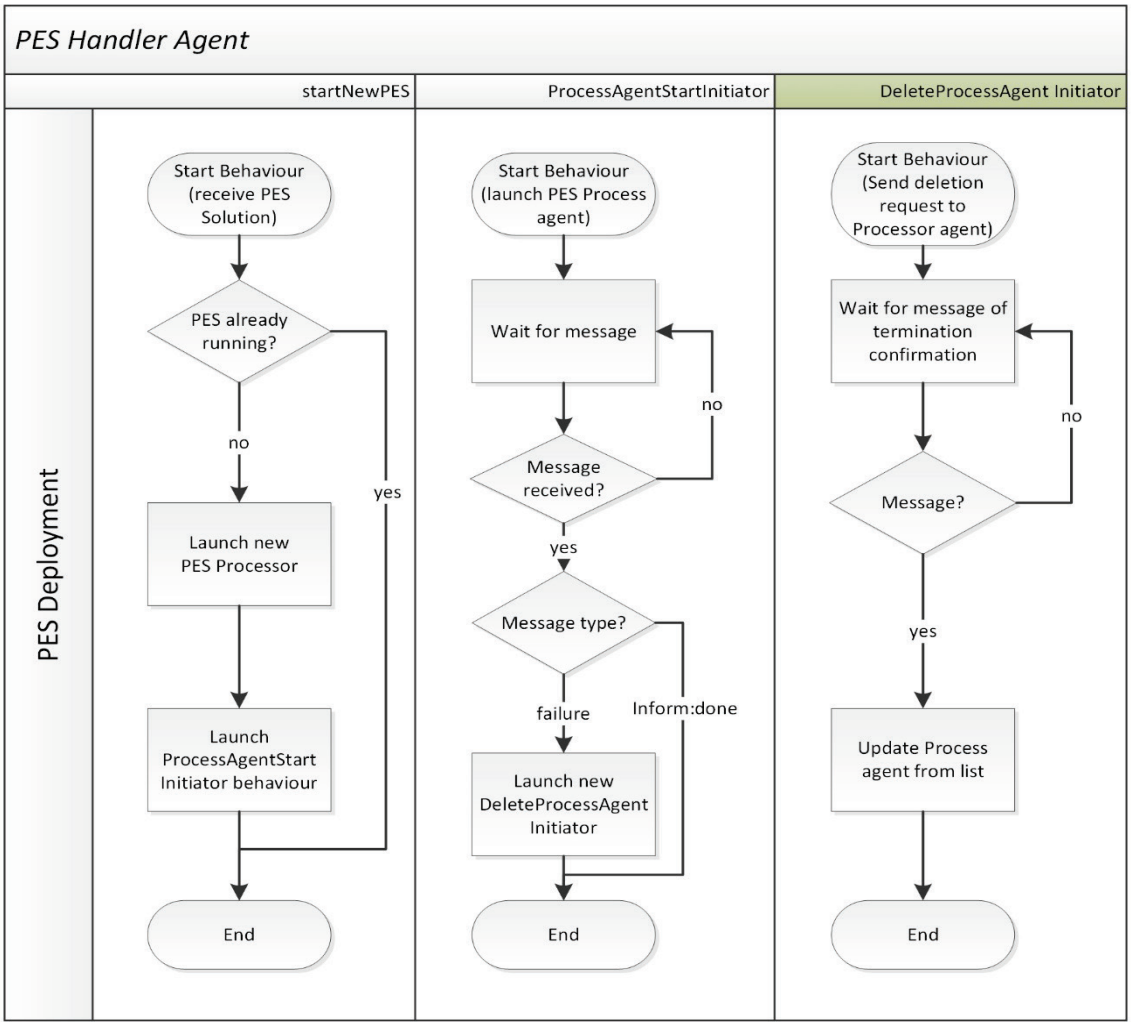

Figure 4.25 - PES Handler agent behaviours related to the PES Deployment

4.2.4.2.2. PES Processor Agent

The *PES Processor* agent (see Figure 4.26) aims to work over the *PES Deployable Solution*, by extracting and interpreting the information that is used to validate the PES consistency and further launch the necessary *Runtime Service agents* in accordance to the selected services and their designed *Service Composition*. For this, this agent is adopted of several fields and methods that provide the registry of information in use, as well as the necessary behaviours (depicted in Figure 4.27) that provide the logic for launching and handling the *Runtime Service agents* needed for the PES Execution.

Figure 4.26 - PES Processor Agent class representation

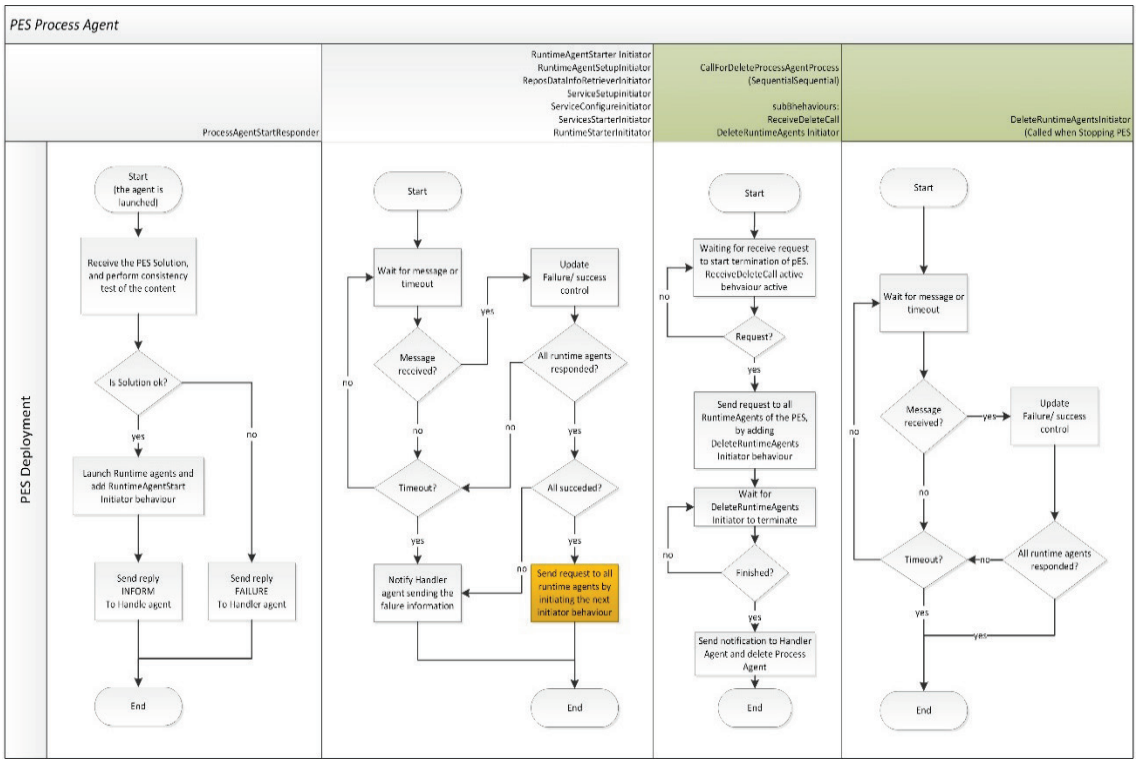

Figure 4.27 - PES Processor agent behaviours realted to the PES Deployment

The first behaviour to be activated is the *ProcessAgentStartResponder* behaviour, which relies on the *FIPA Request Protocol* as Participant, at the time the agent receives a *Request* message with the respective content (ontology match) that enables to identify and trigger this behaviour. The workflow of this behaviour consists on assigning the *PES Deployable Solution* to the field *PES*,

and from the information extracted and created during the *Validation and Consistency process*, the *Service Composition Configuration* is separated and assigned to the *compConfig* field, while the others are kept in an Array List named *configs*. Also, the HashMap *runningServices Map* is assigned with the references to the Services information designed in the PES Development and extracted from the *Service Composition Configuration*. Along this process, several conditions need to be met in order to continue forward without failure, such as:

- From the *hasPESConfigurations* Array of the *PES Deployable Solution*, which is extracted, there's the need of:
    - Exist one, and only one, *Service Composition Configuration* object
    - At least one *PESConfiguration* associated with a *Deployable Service*
- From the *RunningServices* Array extracted the *Service Composition Configuration*:
    - There is the need to exist a match between each of the *RunningService* objects and the *PESConfigurations*

If case any of these (and others) conditions fails, the agent will reply to the *Broker Handler agent* with a *failure* message and will keep on hold for any command (e.g. to start the deletion process). In case no error occurred while interpreting the *PES Deployable Solution*, then all parameters were extracted and interpreted correctly from it, meaning that for each *PES Configuration* a new *Runtime Service Agent* is launched, an *inform* is sent as reply to the *Broker Handler agent* and the *RuntimeAgentStarterInitiator* is activated, before terminating the current behaviour.

The *RuntimeAgentStarterInitiator* behaviour is the first of several similar behaviours that are **sequentially** triggered and where this agent acts as synchronizer of the *Runtime Service agents* involved. The sequence relates to the workflow performed by the *Runtime Service agents* as they perform the tasks for allocating, setup and start the resources (*Deployable Services* and *Proseco Repositories*) as is explained in more detail on the next section.

The synchronization is started by sending a message to all the *Runtime Service agents*, and evolves by collecting all the replies or until a specified timeout is reached. When the timeout is reached, or if any *Runtime Service agent* replies with a *failure* notice, then the system will move into recovery mode, where it can retry the procedure or, if no recovery is possible, will terminate the PES execution. Otherwise, the next behaviour is activated until the last is concluded, meaning that everything was setup correctly and that the PES execution started with no errors.

Also, when the process to stop the associated PES is triggered, the *CallForDeleteProcessAgent* behaviour is awaken, which by its turn will send requests to the *Runtime Service agents* to proceed to their termination process. This is also a synchronized task, given to the hierarchical structure

of the agents. Once the process for deletion is started the behaviour will wait for all replies from the *Runtime Service agents*, and then, a notification is sent to the *Broker Handler agent* and finally the *PES Processor agent* is terminated as well.

4.2.4.2.3. Runtime Service Agent

The last agent that steps into action is the *Runtime Service agent*, starting by receiving all the relevant information dedicated to a certain resource (*PES Deployable Service*) and to the system that are needed to interact with the respective resource, allowing the preparation for the PES execution. The agent Java class is represented on Figure 4.28, where is can be identified the fields that are automatically assigned when object is created: the reference to the *PES Processor agent* that launched this agent, *processAgent*; the *Broker Notification Service* information containing the URL, *notifierInfo*, and; the *PESRunningService service* that contains the information from the resource in use. From the *PESRunningService service*, several other information is extracted and assigned to respective fields from the agent, such as: the service identifier, *service_Id*; the *PESConfiguration, configuration, or*; the *HashMap* with the data flow specifications, flowSpecs.

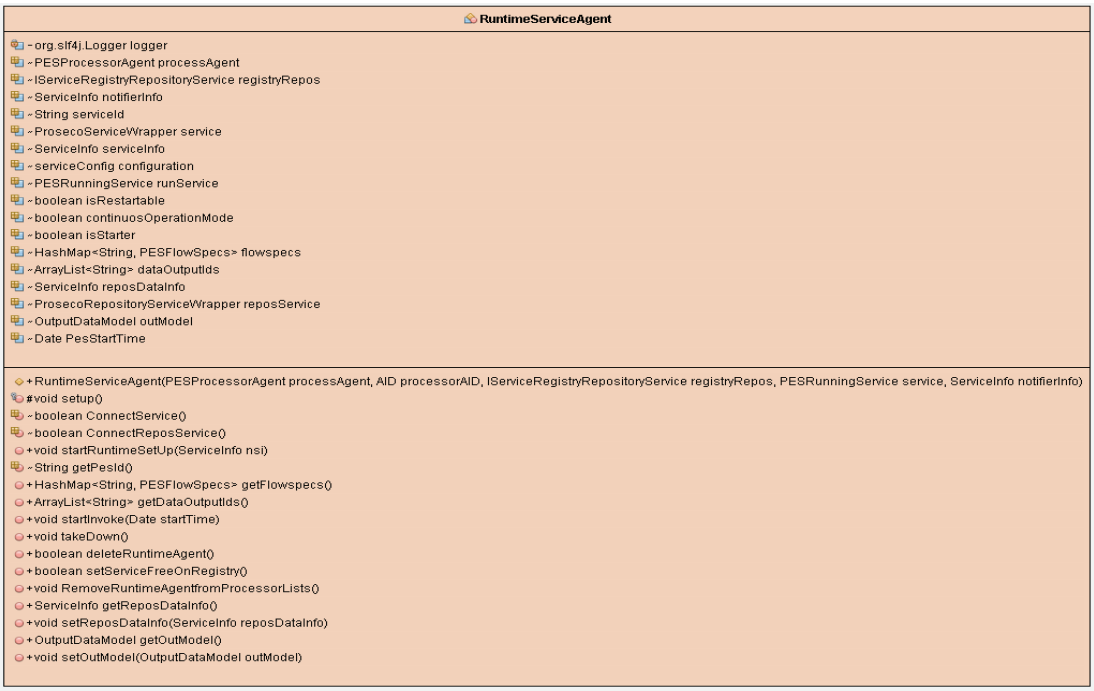

Figure 4.28 - Runtime Service agent class representation

For the *Validation & Setup phase* of the PES deployment, the *Runtime Service agent* is provided of a set behaviours that aim to automatically perform the allocation and setup of the resources to be used for a given PES, and which are related to the workflow of the resources (*PES Deployable Services* and associated *Proseco Repositories*) during this phase - which was presented in section 3.3.3.1.2 -, and also directly related to the *PES Processor agent* synchronization behaviours

presented in section 4.2.4.2.2, (more explicitly in Figure 4.27). The *Runtime Service agent* behaviours depicted in Figure 4.29, were developed for achieving this purpose. All of them are triggered by the *PES Processor agent*, at the appropriated time and reply with either *inform* or *failure*. Every *Runtime Service agent* must successfully complete each task in order to the system goes through all the steps of the resources setup, enabling the start of the PES Execution.

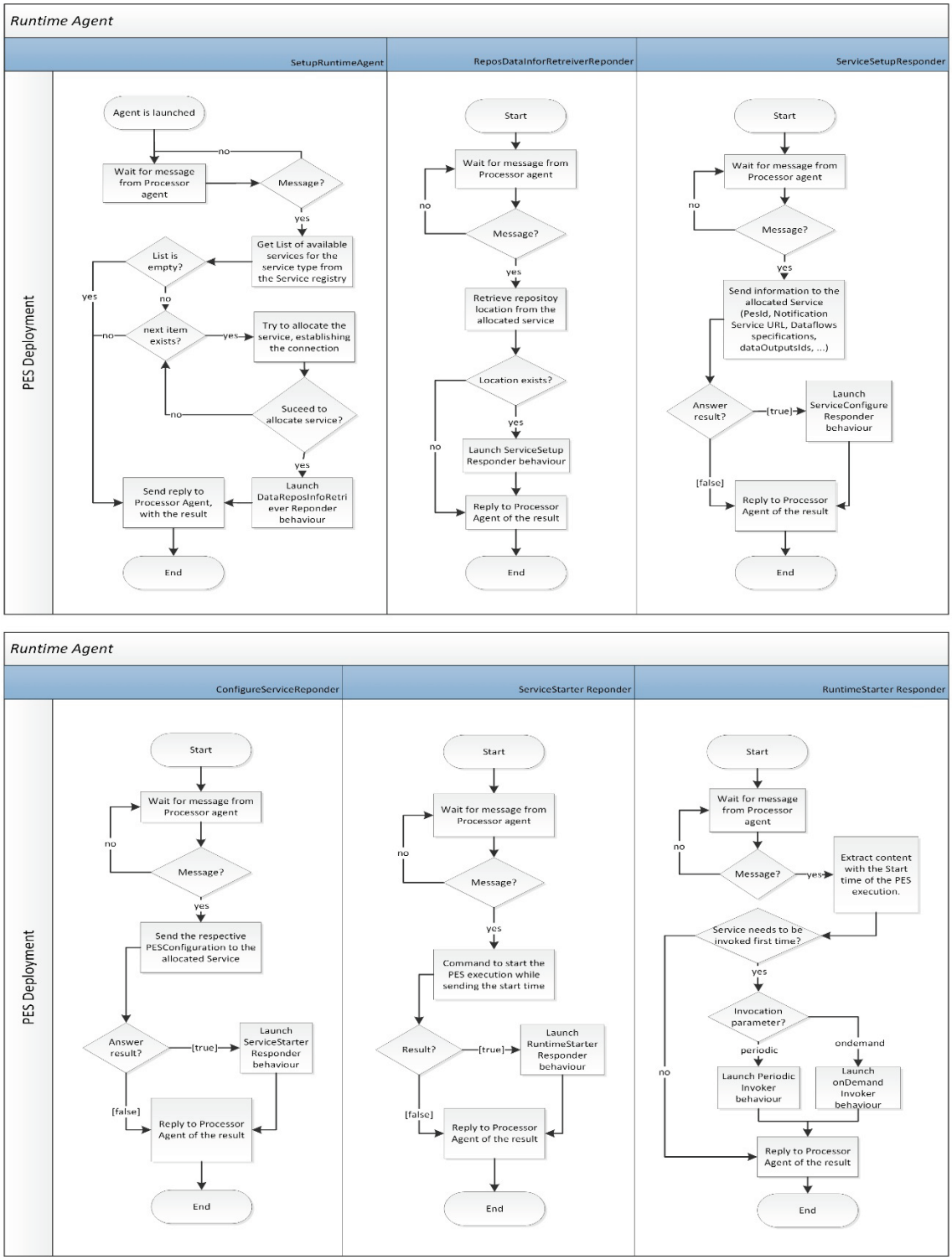

Figure 4.29 - Runtime Service agent behaviours in relation to the PES Setup phase

- *SetupRuntimeAgent* behaviour: the agent searches for the available services of the respective type (*ServiceInfo* field extracted from the *RunningService* object), by querying the *Service Registry Service*, and when one correspondent free service is found, it's exclusively allocated and assign to the *service* field (Wrapper for the client of the resource endpoint). The resource is now reachable by the agent from this point forward, while the status is updated on the *Service Registry* from free to busy. If an available resource was found, the reply is set to an *inform* and the next behaviour (*ReposDataInfoRetreiverResponder*) is launched and waiting for the wakeup call. Otherwise, if there are none resources of the same type, or if they are all set to *Busy* in the *Service Registry*, then the reply is of the type *failure*.
- *ReposDataInfoRetreiverResponder:* when triggered, the service is invoked (*getReposInfo* method) to reply with the information of the associated *ProSEco Repository*. This information is passed to the *PES Processor agent* in order to passed to the other resources who are destined to connect, according to the data flow specifications from the *Service Composition*. In here, the associated repository client is also assigned to the respective Wrapper to the *Repository Service* endpoint, *reposService*.
- *ServiceSetupResponder:* Both *PES Deployable Service* and associated *Repository Service* are invoked with the *setupRuntimeSpecs* and *setupRepos* methods respectively, and the *Broker Notifications Service* location and respective data flow specifications (*flowSpecs*) are passed to them. Once again, the agent replies to the *PES Processor agent* informing the result of the operation.
- *ConfigureServiceResponder:* The respective *PES Configuration* is passed to the *PES Deployable Service* by invoking the *ConfigureService* method. A reply is received with the result of the configuration process, which is translated to the self reply of the behaviour to the *PES Processor agent*.
- *ServiceStarterResponder*: The *PES Deployable Service* and the associated *ProSEco Repository Service* are commanded to start the PES execution, while a start time is acknowledged and assign to the *startTime* Timestamp.
- *RuntimeStarterResponder*: Although the PES has already entered the *Execution stage*, and in case the respective *PES Deployable Service* is involved as receiver of any data flow according to the *Service Composition*, it is the agent that is responsible for commanding the *Deployable Service*, at the right time, to make the first request of data (*InvokeForData* method) for each defined data flow. This behaviour proceeds according to the specifications provided in the *flowSpecs* data structure and further activate other behaviours (such as the *Periodicinvoker* and the *OnDemandInvoker*) accordingly, as will be seen on the next section dedicated to the *PES Execution phase*.

At this point, if no failure and attempt of recovery have occurred, the *PES Execution phase* is already on move, as all the resources have been totally configured and are already in the process of producing results and perform the data exchanges which were defined on the PES Development.

4.2.4.3. PES Execution

It has been already stated that at the *PES Execution phase*, the agent system takes a minor role on the execution, since the resources are able to perform execution choreographically without the intervention from the system. The agent system is then raised into a monitoring system that keeps track of the events related to the resources by receiving *Notifications* through the *Broker Notification Service* endpoint. However, as exception to this, among the features that JADE provides, exists the possibility to set behaviours to wake up at scheduled dates (*WakerBehaviours*). This was found most useful to accomplish some of the time specifications inherent to the *Service Composition* of the PES, as the agent system may perform as orchestrator of the data flows between the resources, under specific conditions.

4.2.4.3.1. Agent System as PES Orchestrator

From the *RuntimeStarterResponder* behaviour seen in the section 4.2.4.2.3, it was stated that for each defined data flow that the resource stands as receiver (specified in the *flowSpecs*), the agent is responsible to invoke the service (*runtimeInvoke* method) the first time it is specified. By invoking this method and passing the respective flow identifier (*flow_Id*), the *PES Deployable Service* will start the data extraction procedure seen on section 4.2.3.3.

According to the *Service Composition*, there are two type of data flow: *Periodic*, where a period has been specified for the data exchange, and; *OnDemand*, where only the time of the first data exchange is defined, and the own service is responsible for starting any other future transactions. According to this, two behaviours have been implemented: the *OnDemandInvoker* behaviour and the *Periodicinvoker* behaviour.

The first will only be activated at the specified time of the first invocation, since once the service replies with the result of the operation, it takes charge of the future calls for extracting data. As for the latest, the behaviour itself makes use of the defined *period* in respect to the data flow, to automatically launch and schedule the next *PeriodicInvoker*.

Both behaviours are wakened at the specified time by the system, and automatically invoke the service (*runtimeInvoke*) while passing the respective *flow_Id*. If a failure notice is received on the reply, a message is sent to the *PES Processor agent*, so it may proceed to any recovery for the

problem. Finally, in the case of the *PeriodicInvoker*, the wake-up time of the current behaviour is added to the *period* retrieved from the related *flowSpec* so that the date of the next invocation is found. Then a new *PeriodInvoker* behaviour is launched with this new date as parameter.

On Figure 4.30, the behaviours developed for the agent, regarding the PES Execution are seen.

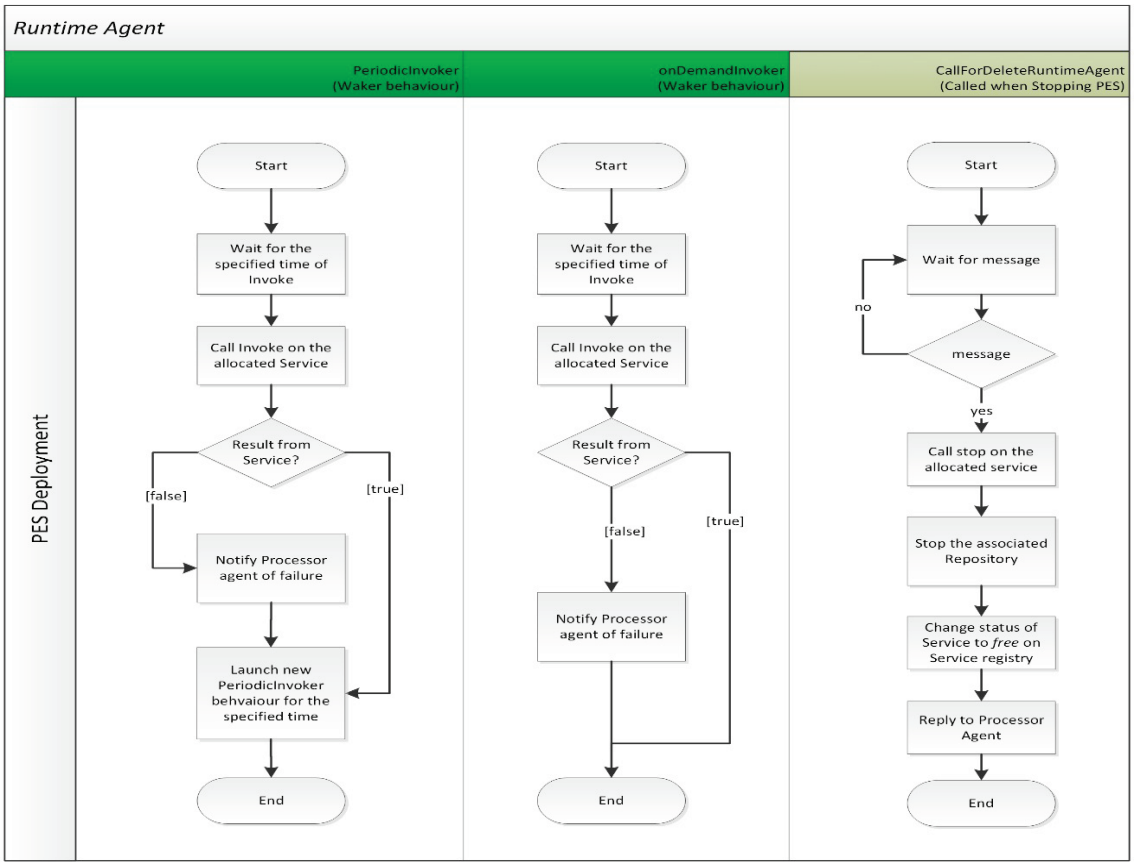

Figure 4.30 - Runtime Service agent behaviours in relation to the PES Execution phase

Also on Figure 4.30, the *CallForDeleteRuntimeAgent* behaviour is constantly waiting until a request to terminate the agent is received (this behaviour is launched and waiting to be triggered since the agent is launched). In this case, the agent will command both *PES Deployable Service* and *Repository Service* to stop the execution, and the *Service Registry Service* will update the status to *free.* Only then, a reply message is sent to the *PES Processor agent* and the lifecycle of the agent is terminated.

4.2.4.3.2. PES Execution Monitoring

The PES Execution is much centred on a choreography performed by the allocated resources (excluding the case seen on the previous section), where the agent system has no intervention on the defined workflow. From the point of view of the performance, this decentralized architecture is more efficient since the resources don't need to have outsider contact with any other

components while acting on the PES execution tasks. However, as a counter point, since the system does not take part on the PES Execution, it isn't able to directly keep track of the events that are happening.

For attributing the system with the ability of monitoring the PES execution, an internal endpoint connected to the agent system is present on the infrastructure – *Broker Notifications Service*. This service, represented in Listing 4, allows the system to receive *BrokerNotifications* from the resources in use on any PES Deployment, through a web method that can be invoked at any time, since the resources are acknowledged with the service location during the *Validation & Setup phase* of the PES deployment.

Listing 4 – IBrokerNotificationsService

```
1.  @WebService(name = "BrokerNotificationsService", targetNamespace = "http://pros
    eco-project.eu/")
2.  @SOAPBinding(style = SOAPBinding.Style.DOCUMENT)
3.
4.  public interface IBrokerNotificationsService extends IProsecoPrimitiveService {

5.
6.      @WebMethod(operationName = "sendNotification")
7.      public void notifyBroker(@WebParam(name = "brokerNotification") BrokerNotif
    ication notification) throws ProsecoFault;
8.  }
```

The *BrokerNotification* class is Java class (see Figure 4.31 a) with fields to include the respective identifiers of the PES, Service and data flow (if necessary) of the sender resource. These identifiers allow the agent system to identify the resource and, consequently, the respective agent that is responsible for it.

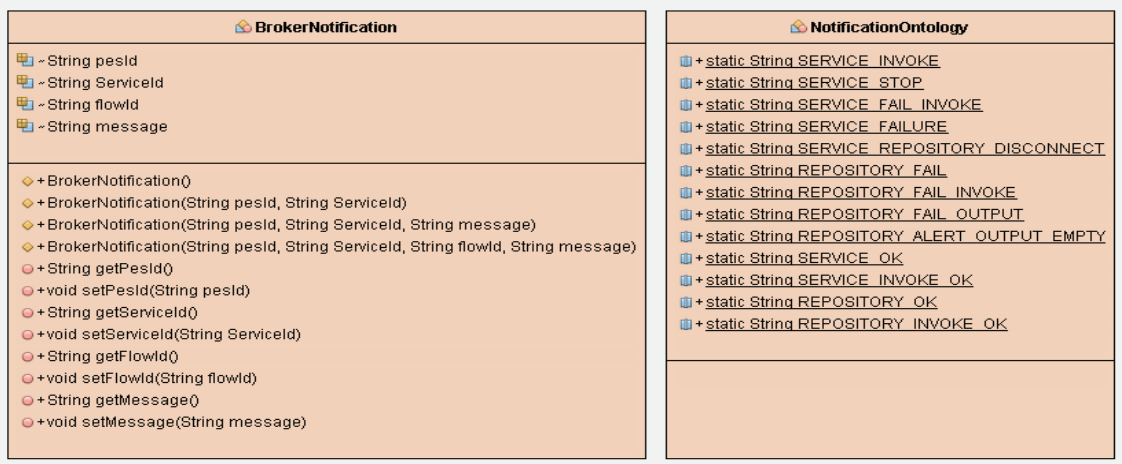

Figure 4.31 - a) BrokerNotification class representation b) Ontology used on BrokerNotifications

The resources are adopted with template methods to automatically create and assign the fields with the correct information, and therefore, may be incorporated during the development.

Also, a predefined ontology (see Figure 4.31 b) is available on the system, already containing several messages to be included on the *message* field that are capable of being understood by the agent system upon the receival of *BrokerBotifications*.

As an example, the developer of the service may determine that some type of malfunction that may occur should force the system to stop the PES Execution. In this case, on the developed code that follows, the developer may include the creation of the *BrokerNotification*, assign the SERVICE_STOP ontology to the *message*, and finally send the *BrokerNotification* using the URL of the *Broker Notifications Service* supplied during the *Validation & Setup phase*. The service forwards the *BrokerNotification* to the *Broker Handler agent*, which is prepared to interpret it. In this example, the *PES Processor Agent* whit the same identifier as the *Pes_Id*, is commanded to start the deletion process.

4.2.4.4. Service Broker User Interface

Noticing that the Service Broker agent system supplies a complete set of functionalities to the PES Deployment platform backend, it was urged the need to provide the users of the platform with a dedicated frontend to visualize and interact with the platform. In this sense, a *Service Broker User Interface* was developed, where it is possible for the users to:

- acknowledge the status of the platform elements as well as of the deployed PES, through graphic visualization
- activate the termination of a given PES deployed and running on the platform

The *Service Broker User Interface* is built as a Web application that provides connectivity to the *Service Broker Service*, and further invocation of methods for completion of the above-mentioned features.

As presented in the Service Broker architecture in section 3.3.2.5, there is a data structure – *UI Elements* - dedicated to store information of the deployed PES and of the platform components, to be further retrieved. This structure is constantly updated when any event occurs on the system, for example, when a new PES is deployed. Also, it is implemented in a Java class named *StatusData*, which contains fields for tracking the information for distinct aspects of the platform:

- *serverStatus*: gathers information of the status of the *Service Broker Service* and of the *Service Registry Service*
- *DeploymentStatus*: to keep the information (identifier, status, etc…) of the deployed PES on this platform
- *AvailableServices*: consists on replicating the resources registered on the *Service Registry*

- *Alerts*: Provides a List of alert messages that were created during the lifecycle of the platform, regarding the PES being deployed and to the platform itself
- *Notifications*: a List of Notifications that were created during the lifecycle of the platform, also regarding the PES being deployed and to the platform itself

The *StatusData* class was developed for this purpose by following the proposed ontological architecture presented in Figure 3.16 from section 3.3.2.5.2, and is represented on Figure 4.32.

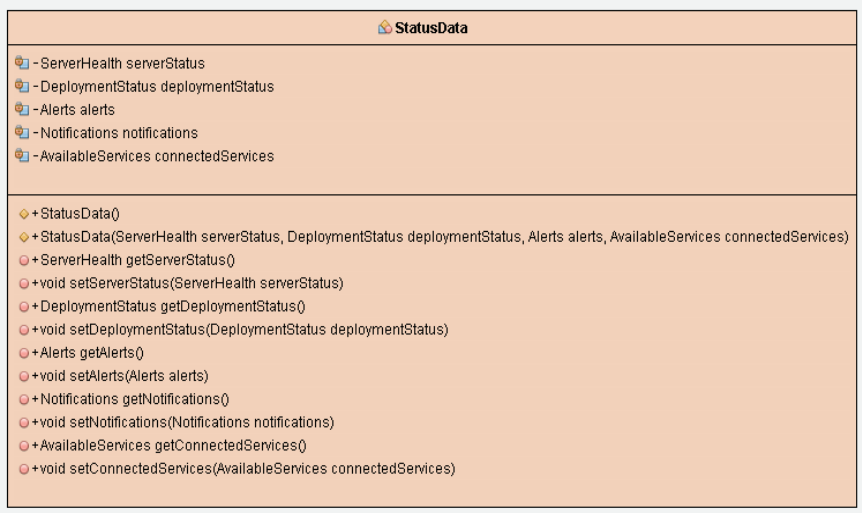

Figure 4.32 - Status Data class representation

4.2.4.4.1. Platform and PES monitoring

Without going into much detail, the operation implemented on the *Service Broker UI* consists on connecting to the *Service Broker Service* and invoke the *getStatus* method. This method collects the current entries of the *StatusData* object and sends them on the reply to the call, where they are interpreted and presented to the user on the GUI dashboard, as exemplified on Figure 4.33:

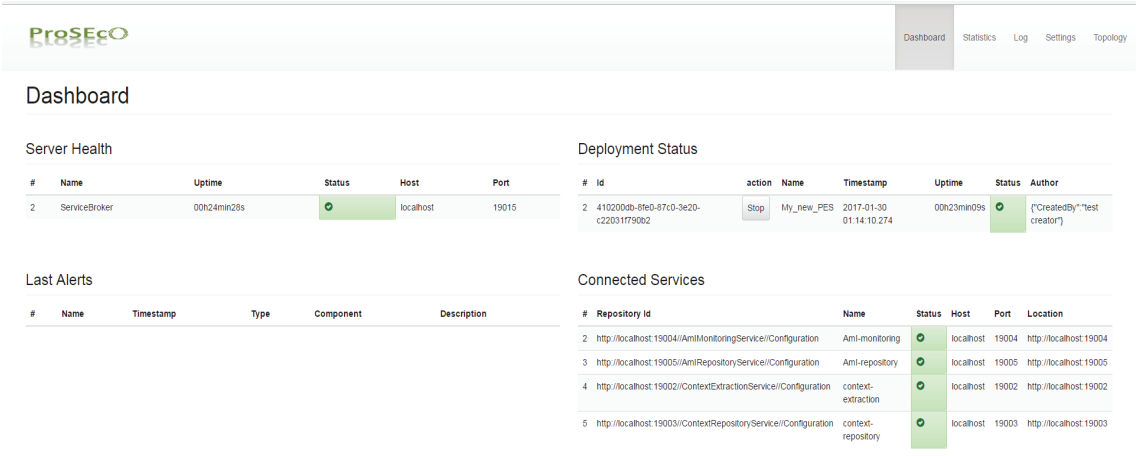

Figure 4.33 - Service Broker UI dashboard

4.2.4.4.2. User interaction for stopping a PES

In the example of Figure 4.33, under the *Deployment Status* it's presented the information of a PES in execution: identifier *id*, name *Name*; startTime of execution *Timestamp*; time since the execution starter *Uptime*; status of deployment *Status* and; name of the author of the PES *Author.* Beside this, it's visible under the *Action* tab, a button that enables the user to trigger the stoppage process of the respective PES.

From the side of the Web application, the process develops by acquiring the respective PES identifier (*PES_Id*) and invoking the *Service Broker Service* with the *stopPES* method, while passing the identifier as parameter, as specified in pseudo-code on Algorithm 8:

Require: Connection to Service Broker Service *client*, Identifier of a Running PES *Pes_Id*
Ensure: boolean value
    Initialization;
        Get *Pes_Id* of the PES
        if *client* endpoint can be pinged **then**
            Invoke *stopPES* method from *client* with **args** *Pes_Id*
        else
            return *false*;
    return *true*;

Algorithm 8 – Service Broker UI *stopPES* method activity

To completely visualize the operation, looking from the platform side at the time the method is invoked, the system reaction is to forward the call from the *Service Broker Service* into the agent system, more specifically into *Broker Handler agent*. The *PES_Id* passed as argument of the method is used to consult the *processorAgentsMap* in order to find the *PES Processor Agent*, that is taking over the PES Execution. Once it is found, the *Broker Handler agent* activates the termination process (seen on Figure 4.25) by waking up the *DeleteProcessAgent Initiator* behaviour, which commands the *PES Processor Agent* to start the deletion process (see *CallForDeleteProcessorAgent* behaviour on Figure 4.27) and consequently activating the deletion process of the *Runtime Service agent* (*CallForDeleteRuntimeAgent* behaviour seen on Figure 4.30) that were created in dedication to this respective PES.

5

Prototype Validation

In this chapter, it will be showed the running application that resulted in the full prototype, which incorporates the presented development of the agent system based on the proposed architecture. The general aspect and features are presented and further exemplified through the presentation of a Business Case covering a set of test scenarios, from an Industrial partner, which was applied in a real technical environment and that serves as one of the demonstrators for ProSeco project.

The first section details the running environment of the system while receiving and executing PES, covering up both Validation & Setup and PES Execution phases of the PES Deployment.

The second one presents the application of the Tests Scenarios and the consequent results that validates the concept presented on this dissertation.

5.1. Runtime Environment

The ProSEco Deployment platform full prototype is delivered as a Java application that provides a SOA framework that enables the users to execute PES, using services that may be distributed

around a network. The application can be settled in a machine as a resource provider (Deployer that hosts the services to be used), or as the core system that integrates the Service Registry and the Service Broker modules. As so, the Starter User Interface is presented when the application is launched, where the user may choose for one of presented options, as seen on Figure 5.1.

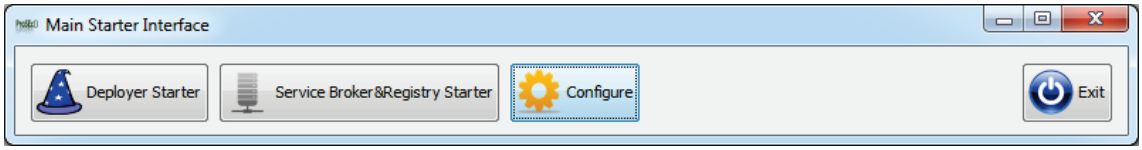

Figure 5.1 - Proseco Deployment Platform Start User Interface

By selecting the Service Broker & Service Registry starter, which is the most relevant for this work, the components are launched in the background, however the application offers some GUIs to be accessed, so that the user may consult some of the features offered by these components.

The Service Registry Main User Interface (see Figure 5.2) contains a log area, where all the events are textually presented:

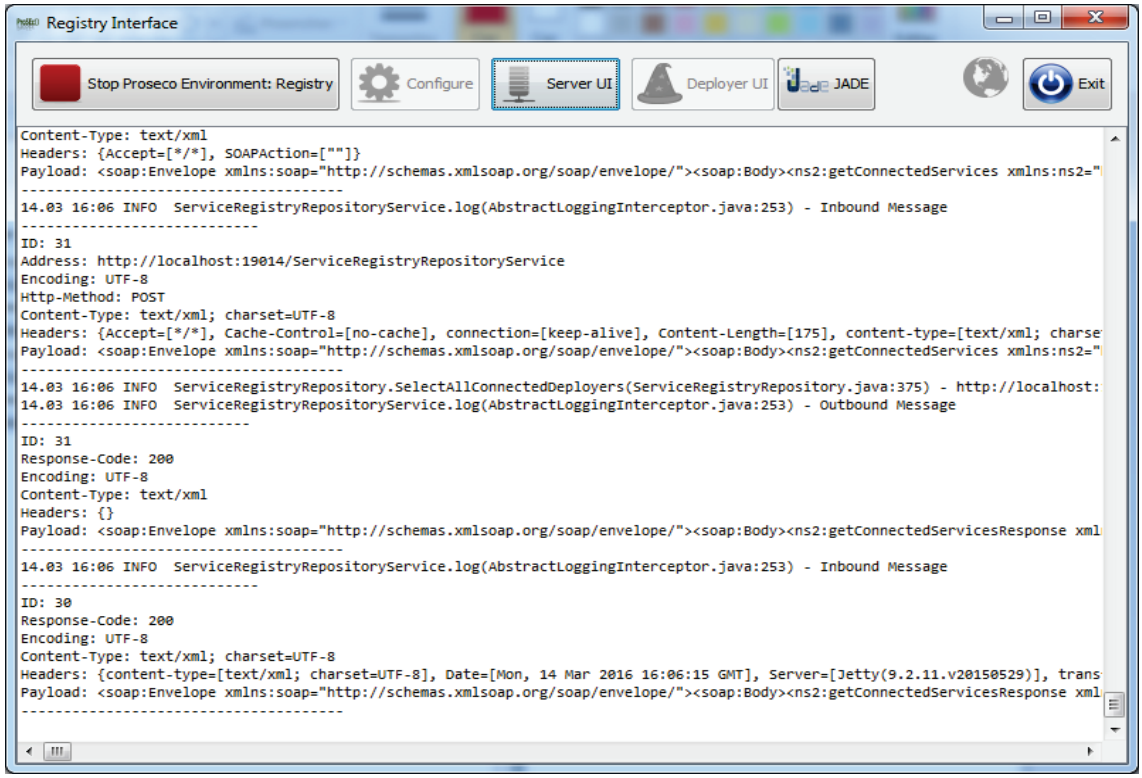

Figure 5.2 - Service Registry Main User Interface

Also, the user may consult the Service Registry, where the resources offered by the Deployers who have registered in this Registry are shown. On Figure 5.3, the Service Registry shows several services hosted by one Deployer that proceeded to the registration on this platform. The resources information, such as the URL, type of service and the availability status is visible.

It was made use of JADE GUI, where it can be observed the agent system in detail, which turned to be very useful during the development of the Service Broker. On Figure 5.4, the main JADE User Interface is presented after the platform is launched, where is visible the presence of the *Broker Handler agent*, which supports the idea of it being launched at the system start-up.

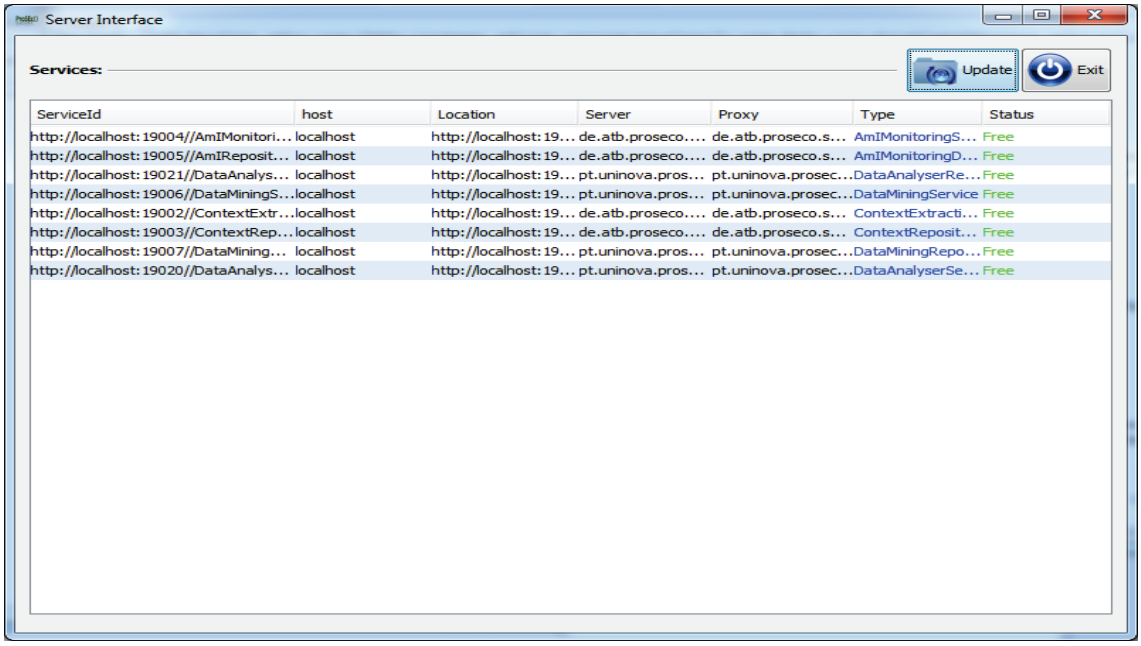

Figure 5.3 - Service Registry table

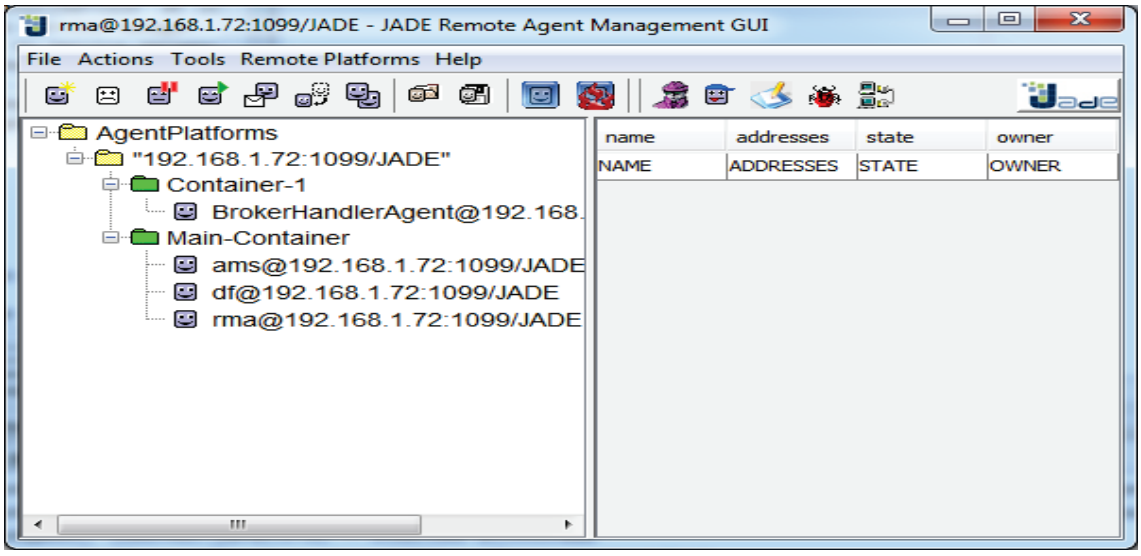

Figure 5.4 - JADE User Interface

5.1.1. PES Deployment show case

In this section, a simple case is presented, highlighting the key features of the implemented prototype described along this chapter, according to the proposed architecture referenced in Chapter 3:

- *PES description*: introspection of the key PES elements
- *PES Deployment Setup*: receiving and launching the PES by the agent system, while using the platform components for proceeding to the resources allocation and setup
- *PES Deployment Execution*: overview of the interoperability between resources and, agent system and other components tasks related to the PES Execution

5.1.1.1. PES description

The considered PES taken for exemplification refers to the *PES Deployable Solution* in Annex I. It was taken a simple PES composed by two services:

- *Specific Data Collector* Service
- *Specific Data Analyser* Service

The *Service Composition* elaborated is presented on Figure 5.5, where a data flow has been specified with the following characteristics:

- The *Specific Data Collector* Service will generate and provide results (sender service)
- The *Specific Data Analyser* Service (receiver service) will invoke the sender service, according to the following rules:
    - The first call for data is set for 10 seconds after the PES Execution starts
    - The receiver service will keep on invoking the sender service periodically, with a *period* of 5 seconds

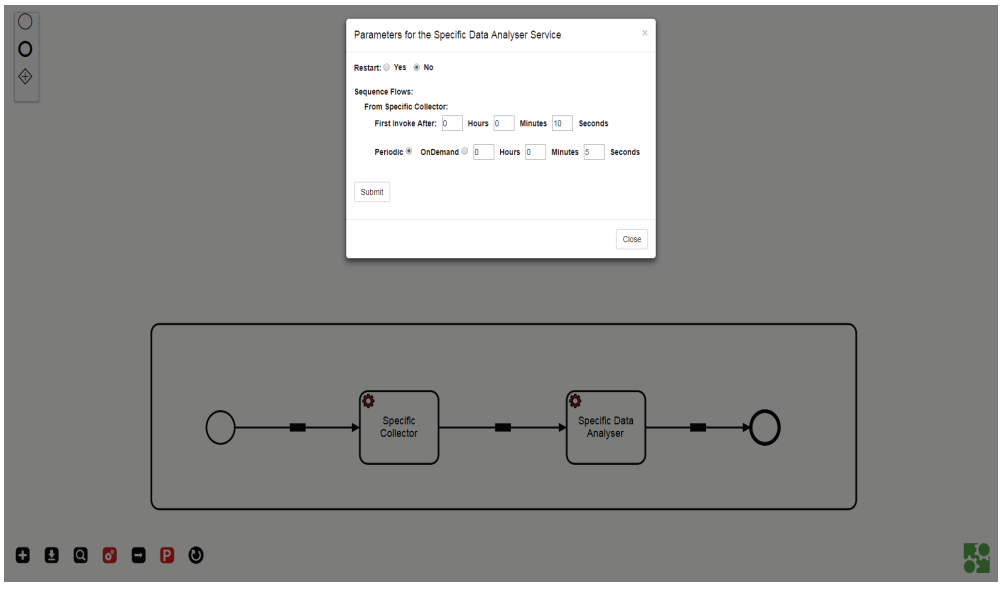

Figure 5.5 - PES Service Composition

It has been specified a data model (as a Java class named *DataColletionModel*, seen in Figure 5.6), which the sender service will provide the results accordingly:

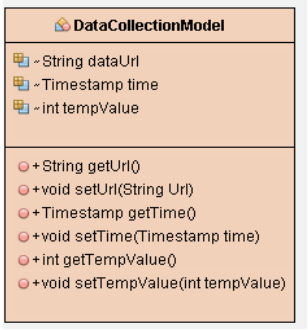

Figure 5.6 - *Data Collection Model* class representation

The following values were assign to the identifiers of the PES, Services and data flow:

PES_Id = "Uninova_Fridge_test_1"
SpecificDataCollector_Id = "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-d6489d2964a4"
SpecificDataAnalyser_Id = "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-644d7adc6bc9"
Flow_Id = "SequenceFlow_1urxz6q"

5.1.1.2. PES Deployment Setup

Once the *Service Broker* invoked to start a new PES Deployment, the request is passed on to the *Broker Handler Agent*, along with the *PES Deployable Solution*. If the request is valid, then the respective agents are created, in accordance to the identifiers:

- One PES Processor agent assign with the *PES_Id* name
- One *Runtime Service agent* for each service part of the PES
  - *SpecificDataCollector_Id*
  - *SpecificDataAnalyser_Id*

The JADE User Interface allows the visualization, in real time, of the agents launched in dedication to this PES, as seen in Figure 5.7:

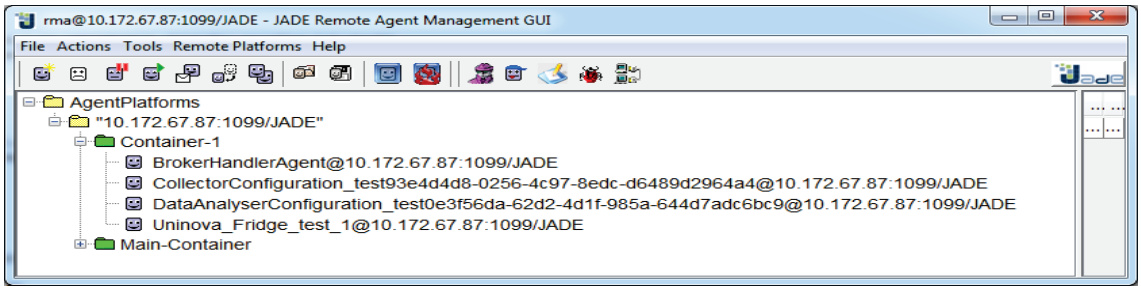

Figure 5.7 - JADE User Interface after launching the agents for a PES

Also, in the *Service Registry*, the resources are set to *busy*, meaning that the agents have been able to allocate them for exclusive use for this PES:

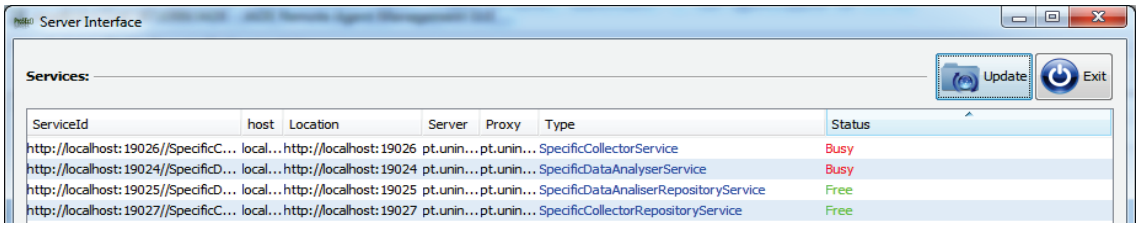

Figure 5.8 - Service Registry table after updated after resources allocation

Finally, if the Setup phase has been successful in every step, the system moves the PES forward into the PES Execution stage.

The setup stage has been tested in a local network, by deploying this PES several times, and extracting the entry date of the request to the *Service Broker Service* and of the PES execution start Time, with the following results presented in Table 5.1:

Table 5.1 Time chart of the PES Validation & Setup stage

Request Time (Service Broker deploy method activated) (hh:mm:ss,SSS)	**PES Execution Start Time (hh:mm:ss,SSS)**	**Time of Setup operation (hh:mm:ss,SSS)**
21:06:24,317	21:06:26,392	0:00:02,075
21:30:27,535	21:30:29,267	0:00:01,732
21:37:58,720	21:38:01,419	0:00:02,699
22:18:48,397	22:18:50,506	0:00:02,109
22:39:01,469	22:39:03,528	0:00:02,059
Mean Value		**2,135 seconds**

5.1.1.3. PES Deployment Execution

For the given PES, the information extracted from the Logger has been taken to prove the concretization of the proposed features of the system:

- Accomplishment of the *Service Composition* specifications by the agent system and the resources
- Interoperability of the resources using the designed data models

For the first, the relevant *Service Registry & Service Broker* logs reporting to the PES Execution start and to the calls made by the agent system for commanding the resources to start the data extraction process, as seen in Figure 5.9 and Figure 5.10:

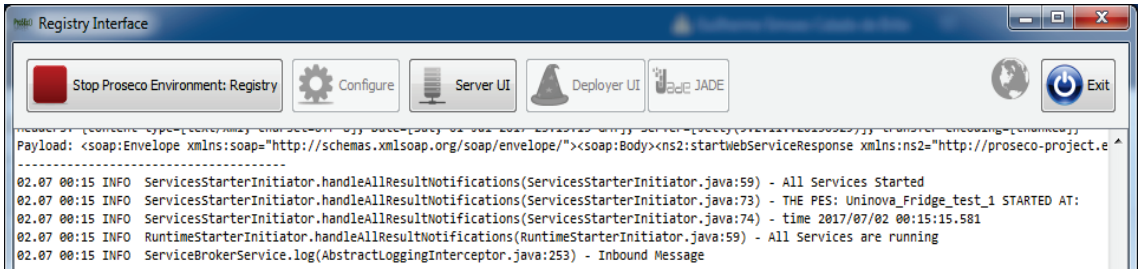

Figure 5.9 - Registry & Broker log area showing the start time of the PES Execution

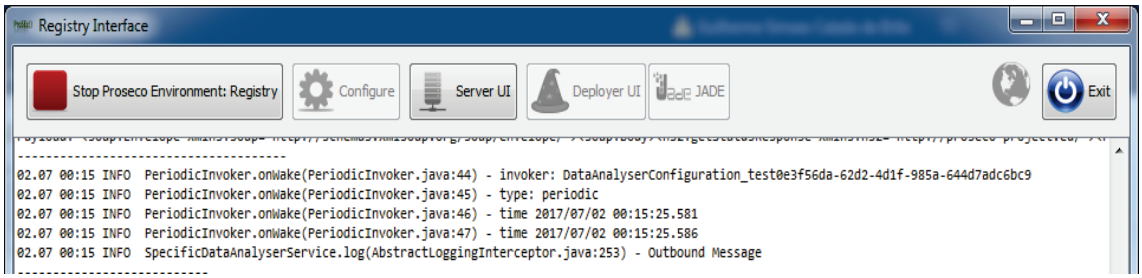

Figure 5.10 - Registry & Broker log area showing the time of the first invoke to the resource

Remembering the data flow specifications, the first invoke to the resource (*DataAnalyser Service*) was set to be triggered 10 seconds after the PES Execution starts, and furthermore, to continue to invoke it every 5 seconds. On Table 5.2, the times settled for the responsible agent behaviour (*PeriodicInvoker* behaviour) to be waken and the times that the method is really invoked are presented. The process encompasses delays of the system related to the activation of the behaviour of the agent, activation of the call method on the service who makes the request and the time the web request itself spends until the Repository is acknowledges the invocation.

Table 5.2 - Time chart of the Periodic Invoker during the PES Execution

	Time programmed (*periodicInvoker*)	**Time on agent (*periodicInvoker*)**	**Time of Invoke (*DataAnalysier Service*)**	**Time of Invoke (*Collector Repository*)**
Execution start time	**00:46:11,544**			
1st Invoke	00:46:21,544	00:46:21,551	00:46:21,613	00:46:21,629
2nd Invoke	00:46:26,544	00:46:26,554	00:46:26,554	00:46:26,569
3rd Invoke	00:46:31,544	00:46:31,547	00:46:31,547	00:46:31,562

	Time programmed (*periodicInvoker*)	Time on agent (*periodicInvoker*)	Time of Invoke (*DataAnalysier Service*)	Time of Invoke (*Collector Repository*)
4th Invoke	00:46:36,544	00:46:36,556	00:46:36,556	00:46:36,572
5th Invoke	00:46:41,544	00:46:41,549	00:46:41,549	00:46:41,565
6th Invoke	00:46:46,544	00:46:46,545	00:46:46,561	00:46:46,561
7th Invoke	00:46:51,544	00:46:51,557	00:46:51,557	00:46:51,572
8th Invoke	00:46:56,544	00:46:56,547	00:46:56,547	00:46:56,563
9th Invoke	00:47:01,544	00:47:01,554	00:47:01,554	00:47:01,570
10th Invoke	00:47:06,544	00:47:06,550	00:47:06,550	00:47:06,566
Delay (Mean value)	-	0.007	0,015	0,029

As for the second point, regarding the interoperability and considering the data model in use for this specific PES which was seen in Figure 5.6, also proceeding to the visualization of the logs, it's visible that the system is able to perform the implemented automatic serialization and further send the requested data as result to the invocation, as presented in Figure 5.11, where the collected results for the 4th and 5th Invoke stated on Table 5.2, respectively, can be seen:

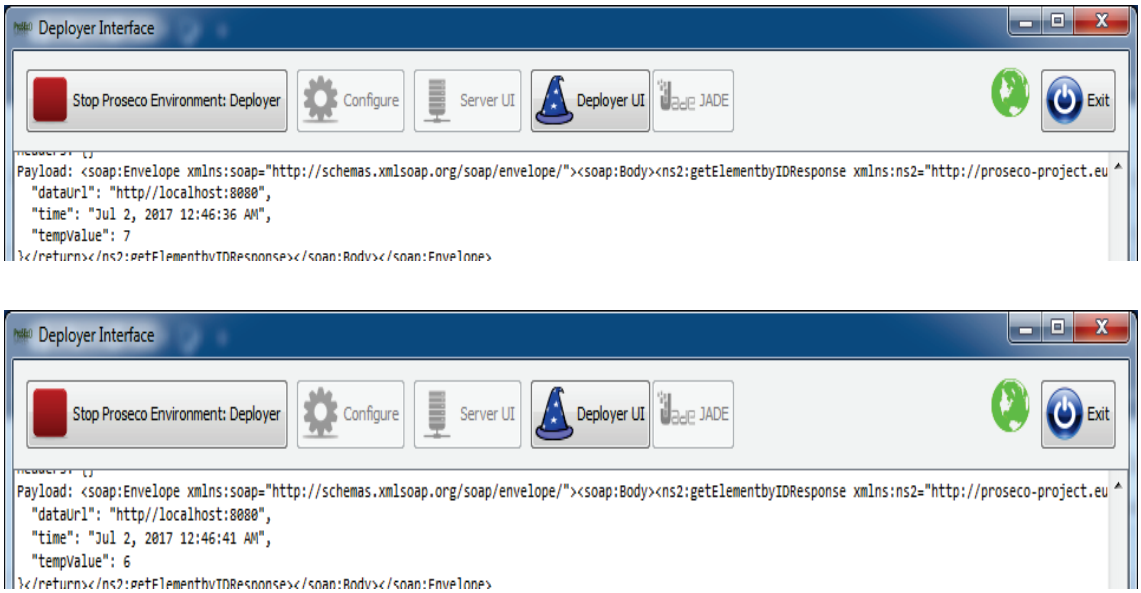

Figure 5.11 – Collected results in JSON format passed as reply to the invoke method

Finally, for complementing the presented implementation of the agent system, a set of logs related to the termination process of the PES is presented on Figure 5.12, where several behaviours are acting and informing of the system activation and completion of the deletion process of the *Runtime Service agents* and of the correspondent *PES Processor agent*.

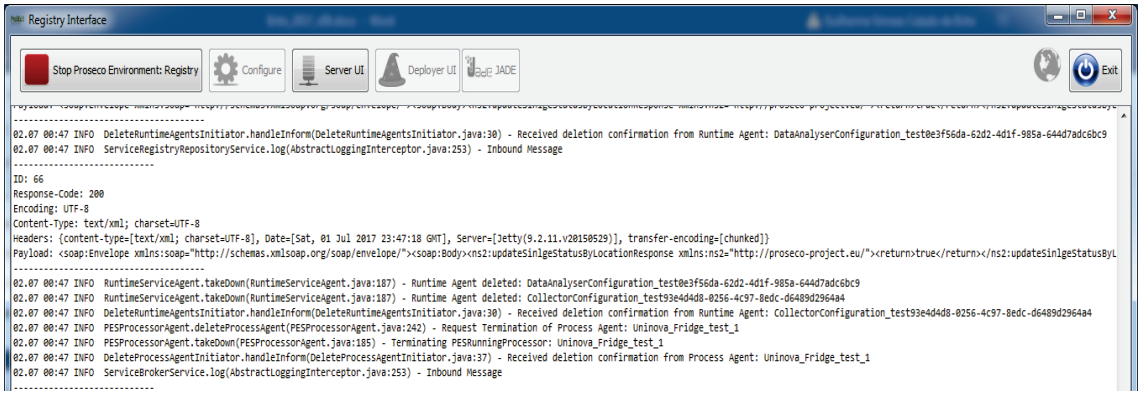

Figure 5.12 - Agents termination activity

5.2. Electrolux application scenario

For validation of the developed prototype, the description and gathered results using the implemented ProSEco infrastructure of one test scenario of the four Business cases, the Electrolux Business Case respectively, is presented.

5.2.1. Business Case and Application Scenario description

The outline of the Business case consists on the continuous monitoring of home appliances as necessary condition to enable the modelling of both consumer and component behaviour along the appliance lifecycle. The full ProSEco solution offers the necessary conditions to collect/extract data from the appliances and the capability of analysing it to find trends and directions for identification of possible failure causes or opportunities for improvements.

The application scenario is geared into modelling the consumer behaviour and further use it for applying a customer Eco-Rating system that leverages the effective use of the home appliances by means of rewarding the customers, and on the other side, to perform adaptive control by optimizing the parameters/variables of the home appliances based on normal use of each customer.

For development and testing of PESs, Electrolux supplied a home appliance (refrigerator) which was connected to ProSEco system, thus creating a real infrastructure:

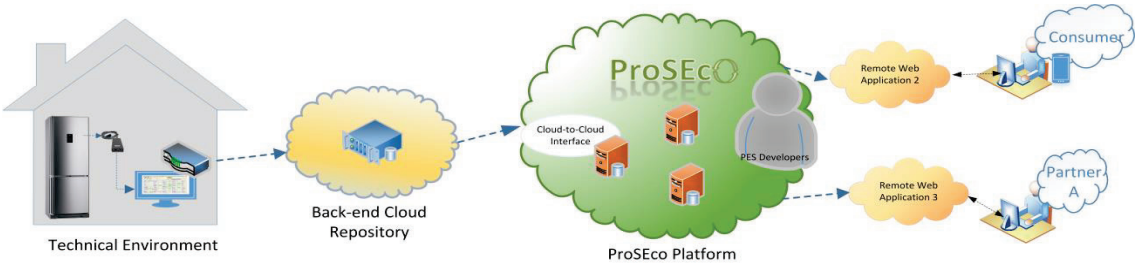

Figure 5.13 - Technical infrastructure in use for Electrolux Business case

With this appliance connected, the conditions for implementing the several test cases of the application scenario were created.

5.2.2. Use Cases Description and Results

The considered Use Cases consists on using ProSEco as a mean to model the consumer behaviour by using the functionalities of ProSEco services. Therefore, two services were found essential to use and compose for developing the PES:

- *AmI Monitoring Service*: for collecting data from the connected appliance
- *Data Mining Service*: for providing the necessary analysis over the monitored data

Regarding the *AmI Monitoring Service*, once the product and associated sensors have been defined in the ProSEco system, it provides the ability to extract the data from the appliance and store it according to the ProSEco ontology system, therefore capable of being used along other components and resources. To notice that the functionality offered by this resource is transversal to the use cases presented.

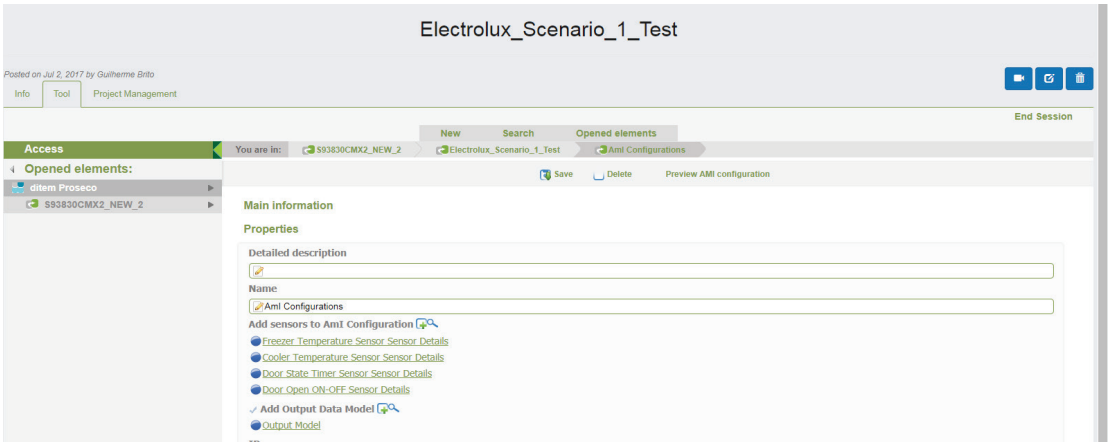

Figure 5.14 - Selecting Product and relevant sensors for the PES (*AmIMonitoring* Selection Tool)

An external service has been created to receive and visually present the data extracted from the *AmI Monitoring Service*.

Figure 5.15 - Visualization Service for extracted data

5.2.2.1. Use Case 1 – Predicting the hours of use of the appliance

The objective of the test is to determine that the refrigerator is (not) used, by modelling the door opening, and therefore predicting the future use of the appliance. The data collected by the *AmIMonitoring Service* is retrieved by the *DataMining Service*, which by its turn, perform the necessary analysis. The *DataMining Service* allows the selection and parameterization of a set of algorithms used for data analysis, beside the inclusion of new ones. The parameterization is done on the PES Development, by means of the *Data Mining* Engineering Tool. On Figure 5.16, the selection form for selecting an algorithm can be seen:

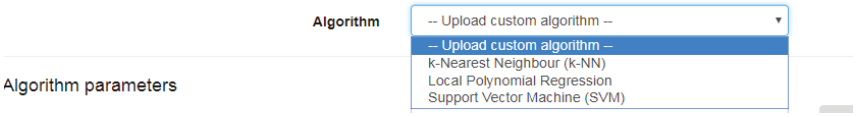

Figure 5.16 - Data Mining Engineering Tool snapshot of the Algorithm selection

5.2.2.1.1. Moving average algorithm

The first algorithm to use is triangular moving average due to its simplicity and needless of computational resources (therefore could be implemented on small processors). As the daily number of meals varies around 4 (in 12 hours actually), the window width was set to 3 hours. The results are shown in Figure 5.17 and Figure 5.18.

Due to the enlarged set of data at disposal, and restriction of the data mining software (RapidMiner) academic license in use, no more than 10.000 records for processing is allowed.

Considering this, for providing the results for this Use Case, as well for the following one, the initial data has been aggregated hourly-based. However, applying the same type of analysis over a larger spectrum, while reducing the sample frequency, the results can be more accurate.

The following charts depict the hours, for approximately one month, that the refrigerator door was opened:

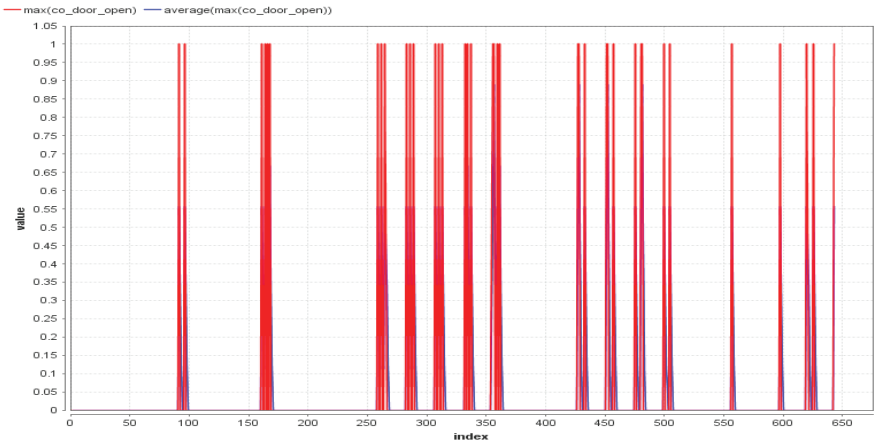

Figure 5.17 - The hours when the refrigerator door is opened (red) and the moving average (blue)

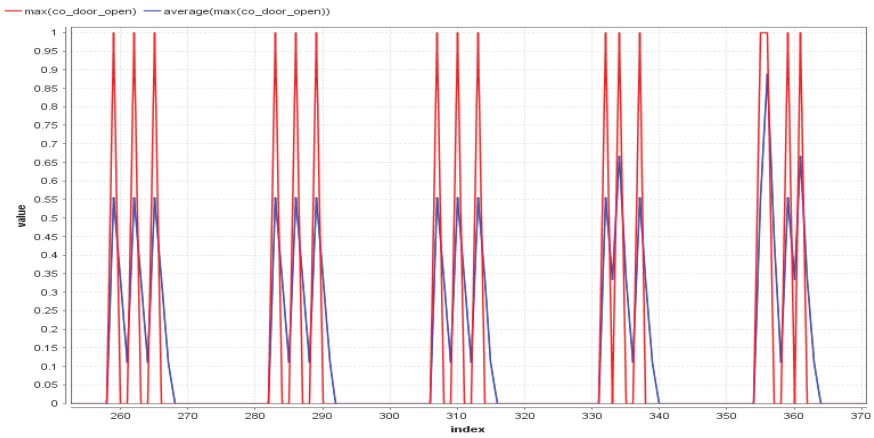

Figure 5.18 - A zoom in the chart depicted in Figure 5.17

It's important to notice that a periodicity occurs, enabling the prevision and consequent model of the behaviour. As the refrigerator is settled on an office room, it can be visualized the cycle of two consecutive days (weekend days), that the door is not opened and also, from Figure 5.18 (zoomed time window of the graphic presented on Figure 5.17), the hours that the door are opened on each day of the week are very similar.

For precision purposes, the values of the moving average (blue) are set with two decimals. However, the value that states that the door is open or close is a Boolean value, and so the floating values are rounded to 0 or 1 (false or true), cutting of the error between the actual values and the

moving average to zero. Adding this to the computational resources for moving average being very small, this was the only algorithm tested in Use case 1.

5.2.2.2. Use Case 2 – Determining the number of door openings per hour

The objective of the test is to determine how many times the door is opened every hour. Once again, the set of data was hourly-based aggregated, and the charts step value (index from X axis), stand for one hour.

5.2.2.2.1. Moving average algorithm

Again, this algorithm is used, this once with an error of 39.33%. A zoom of the results is seen in Figure 5.19.

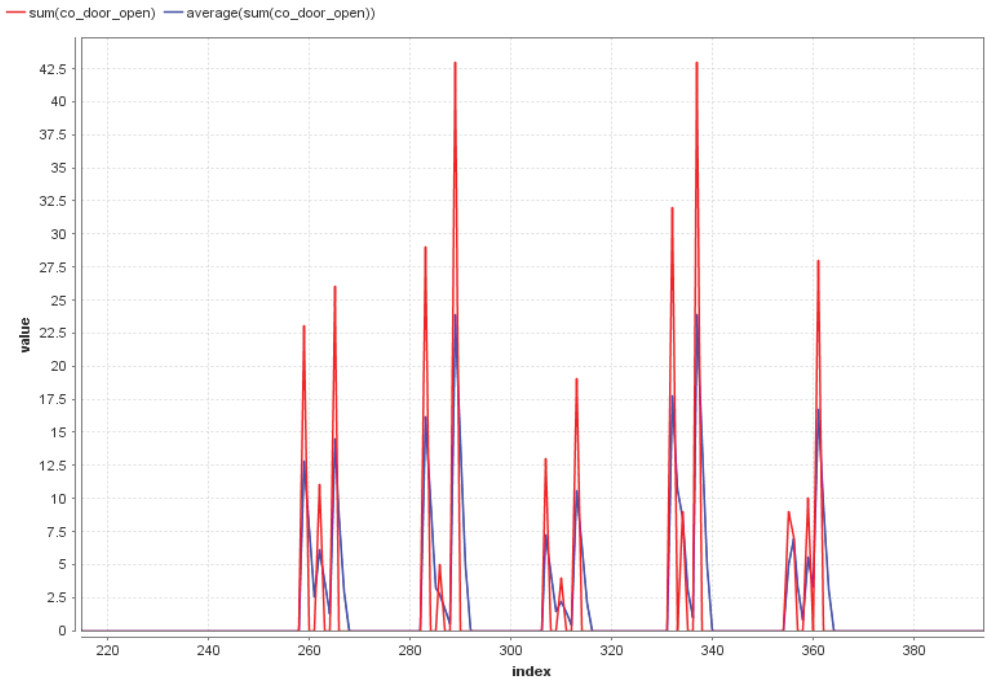

Figure 5.19 - A zoom in the results of the moving average for opening counts

The results show that, although the number of openings predicted for each hour is not completely accurate, the prediction of the hours that the door is opened is pretty much correct. With a more efficient set of data, the associated error can be reduced at the results can be improved.

5.2.2.2.2. k-nearest neighbour

The second algorithm tested is the k-nearest neighbour, with an associated error of 82.87%, but with a good remnant memory over days and even weeks as seen in the blue peaks around index 150 and 250, shown on Figure 5.20.

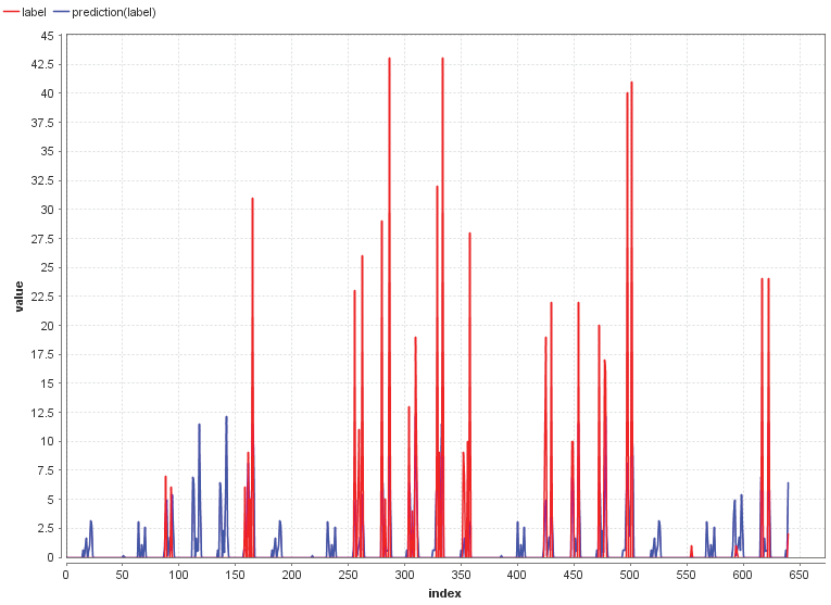

Figure 5.20 - Expected values (red) and predicted ones (blue) using k-NN algorithm

Also in this case, it is important to notice that every real occurrence could be predicted (blue lines underlying the red ones), even that some false positives appear.

5.2.2.2.3. Local Polynomial regression

Finally, the last algorithm is the Local Polynomial regression, which has a very good capability of modelling the number of openings per hour:

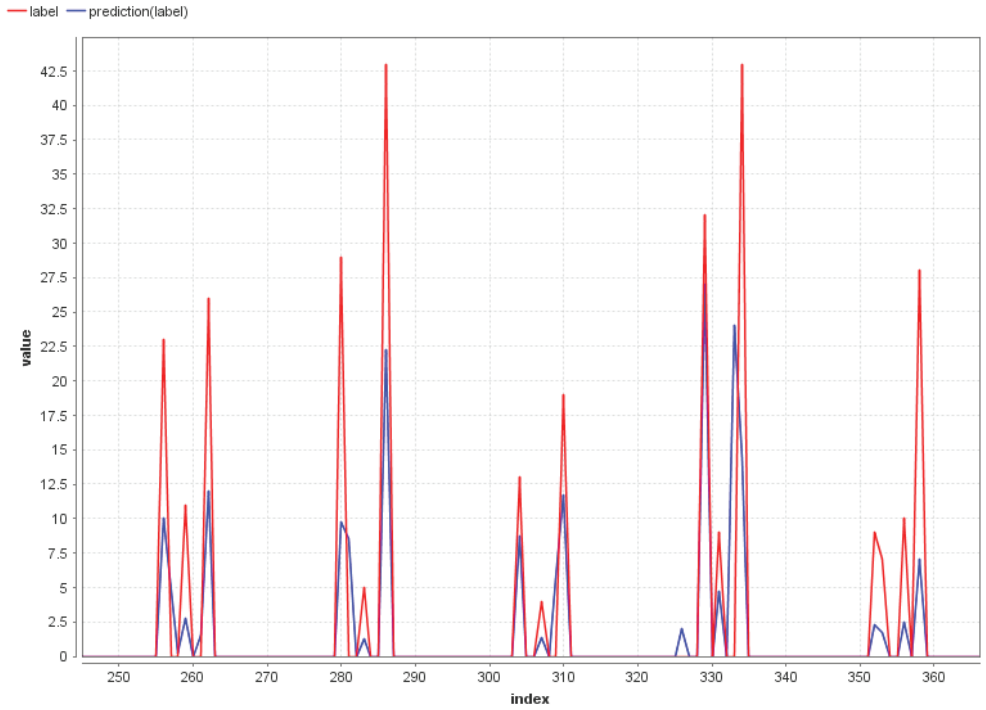

Figure 5.21 - Zoom in some opening counts modelled with local polynomial regression

6

Conclusions and Future Work

Nowadays, the modernization of manufacturing industry is moving towards the introduction of new business models in order to uplift their value, steering from manufacturing and supply of physical product into adding integrated solutions that enhance new functionalities and services as extensions of these products, as well as to the manufacturing processes. Such PESs are applicable to a wide variety of applications such as automotive industry, support to machinery users or maintenance of home appliances, and their concretization is leveraged by the inclusion of cyber-physical features on the physical products and machines. In this sense, the virtualization of such products/processes and associated features enables the idealization and further development of PES, by means of established and possibly enhanced Product-Service Systems. To the information collected from the cyber-physical features, new engineering tools supporting the most advanced technologies in use can be applied such as: Ambient Intelligence and Context Sensitive, Data Mining and specific data analysis features or the application of Eco-driven design techniques. The end target is to be able to combine (and integrate new ones) such engineering services, in order to provide PES that will not only serve to augment the offer of products available to the consumers, but also for enhancement of their own manufacturing enterprise infrastructure

by supporting Product and Service Lifecycle Management integration. The strategic objective of ProSEco project is to provide a novel methodology and comprehensive ICT Solution for the collaborative design of PES.

6.1. Hypothesis Assessment

In the developed work described on this document, it is proven that the proposed architecture of ProSEco is viable for supplying the users the ability to develop PES based on the composition of independent services that interact with each other towards a specified objective.

That said, the imposed vision of ProSEco of using *Service Composition* paradigm, which stands as the mainstream pattern for the realization of Service-Oriented Systems, is self-answering to the first research question of this dissertation. However, creating a full system capable of performing accordingly with the created composition was at some points an extensive work, in the sense that providing the necessary automatized features implies the existence of a full understanding of all the information exchanged between the several components. Nevertheless, being the infrastructure ontology driven, it has eased the overcoming of this obstacles.

As for the second research question, which inquires which are the most viable architecture and technologies for the deployment environment that leverages the execution of the composed PES, it must be considered that development of the *Service Broker* had to fit into an already structured architecture, and imposed to follow the standards in use (such as integration methodology or ontology-based). But, in a more internal perspective, the author was able to specify and apply all the internal structure and intelligence of the module. In here, the concept of *Agents* was considered since the moment several equalities between its characteristics and the ones of the ProSEco system were identified. Multi-Agent systems are ideal, and widely used, to work as detached independent resources or emulating systems.

Supported by the fulfilment of the relevant requirements presented in section 3.2, the proposed solutions are able to accomplish the objectives in a satisfactory way:

- The relation between the *Service Composition* specifications and the developed general features adopted by the Services responsible for the execution of the PES accordingly, provide an easy way for users to define the interoperability of the services, while enhancing the integration of new types of Services, regardless the specific intent.
- As for the *Service Broker* agent-based system, its development, testing and validation using real application scenarios, it can be concluded that it is capable of accomplishing the interpretation of the Service Composition specifications and intervein accordingly with

the necessary resources, providing the automatized mechanisms for handling and management of the executing PES, whilst participating on it if necessary.

- Easy integration of new resources was not affected with the changes promoted by the development of the Service Broker
- The entire Solution has positive response to continuous operation, use of standards, maintainability acceptance, user-friendly and compliance requirements.

Gathering all results, in addition with all the other developed components of ProSEco infrastructure, the main objective of providing a full ICT solution for the development and deployment of PES was successfully employed.

6.2. Challenges and Constraints

Due to familiarization, the use of JADE framework for implementing the specified agents has softened the overall work. However, developing the overall structure and intelligence of Agents from scratch could be considered, possibly leading to an improved efficiency of the agent-based system performance.

Other important decision over the implementation was choosing the system approach to either follow *Orchestration* or *Choreography* as the *Service Composition* pattern. From *Orchestration* point of view, the *Service Broker* as central entity could provide an improved control over the workflow of execution, as he has the knowledge of all the elements and transactions of messages and data. This would benefit the monitoring of the resources since all decision making would stand on the central entity but, from the downside, the *Service Broker* would be overloaded as the orchestration features consume a much higher effort from his side. On the other hand, the *Choreography* perspective loosens the *Service Broker*, therefore increasing its capacity of deploying the incoming PES, but with the cost of performing a non-direct monitorization. The pending factor to opt for *Choreography* was that the resources work with exclusiveness for a given PES at the time, which makes them more suitable to be in charge of the *Service Composition* related tasks while keeping a highly reduced loss of ability to perform their specific tasks.

The biggest challenge that raised along the development relates to the interoperability between the resources, once the deployment system needs to be prepared to accept any defined data exchanges. Enabling the automatization of the interoperability leaded to work on extending the system's ontological model, while adopting the ability to specify simple data models to use over the data flows. This increased the PES Development complexity by forcing the PES Designers to specify these data models in compliance. Moreover, the communication protocol of the APACHE framework used for implementing the services that execute the PES (therefore that perform the

data transactions), was found unable to recognize certain type fields. This had to be work around by inserting an automatized codification (and consequent de-codification) with is used in each time data is passed from one service to the other.

6.3. Future Work

Besides the obvious inclusion of new Services that supply new functionalities for the development of PES, and in regard to the work presented in this dissertation, future efforts should focus on improving and adding features to both *Service Composition* and *Service Broker* agent system.

For the *Service Composition*, one of the improvements considered is to allow the encapsulation of full compositions and use them in the process to create more complex ones, elevating to a higher level of PES, while leveraging reusability. However, it must be considered that any improvement that affects the defined Meta-language created for specifying the interoperability between services may imply extra development on the generic features of the Services, so that they still be able to cope along the PES Execution.

For the services, there is open space for improving the definition of the data model's structures, as for now they are based on previously defined Java classes. Defining a base model structure on which any type of generated data may be generically represented and translated should be considered, as it would benefit users on the development phase since it would decrease the necessary specification work during development phase of PES. For supporting a base model structure, one possibility is to adopt the Service Composition Engineering Tool with such ability, however, this tool would need to be elevated to a higher level of the collaborative environment, becoming the central (or one of the central) module of the Development platform.

On the side of the *Service Broker* agent system, as a monitoring element of the PES execution, it already can identify several failures related to. However, the creation of sophisticated recovery features associated to the monitoring of PES has been left in the open for possibly being introduced in the future.

It is believed by the author of this document, that if there's a dedicated machine for exclusive instantiation of one *Service Broker*, a large number of PES may be controlled by it without a significant cost of performance, as most of the *heavy* tasks are done on the setup phase, and adding the consideration that the number of PES that are being setup are always a minimal fraction of the ones that are being executed. Still, enabling to distribute several *Service Brokers* on connected machines, for scaling up the capacity of the system, and implementing a communication system for this purpose can be considered, if in the future it turns into a requirement.

6.4. **Scientific Contributions**

The work done under the scope of the ProSEco project by the author widely described in this document resulted in scientific contributions that have been published:

- Brito, G., Di Orio, G., & Barata, J. (2017). Orchestrating Loosely Coupled and Distributed Components for Product/Process Servitization. 15th IEEE International Conference on Industrial Informatics, INDIN 2017.
- Lima-Monteiro, P., Brito, G., Dionísio Rocha, A., Ilheu, P., Freire, J., Barata, J., & Cenedese, C. Service-Oriented Architecture to Retrieve and Visualize Data using ProSEco as Data Processing Platform. 15th IEEE International Conference on Industrial Informatics, INDIN 2017.

Bibliography

Babiceanu, Radu F, and F Frank Chen. 2006. "Development and Applications of Holonic Manufacturing Systems: A Survey." *Journal of Intelligent Manufacturing* 17 (1):111–31.

Baines, T S, H W Lightfoot, S Evans, A Neely, R Greenough, J Peppard, R Roy, et al. 2007. "State-of-the-Art in Product-Service Systems." *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 221 (10):1543–52. https://doi.org/10.1243/09544054JEM858.

Barata, Diogo Alfaro Cardoso da Gama. 2015. "Product Extension Services Deployment Platform." http://run.unl.pt/handle/10362/16382.

Barata, José, Pedro Santana, and Mauro Onori. 2006. "Evolvable Assembly Systems: A Development Roadmap." *IFAC Proceedings Volumes* 39 (3):169–74.

Barry, Douglas K. 2003. *Web Services, Service-Oriented Architectures, and Cloud Computing*. Morgan Kaufmann.

Bellwood, Tom, S Capell, L Clement, J Colgrave, MJ Dovey, D Feygin, AHR Kochman, P Macias, M Novotny, and M Paolucci. 2002. "Universal Description, Discovery and Integration Specification (UDDI) 3.0." *Online: Http://uddi. Org/pubs/uddi-v3. 00-Published-20020719. Htm* 10.

Beuren, Fernanda Hänsch, Marcelo Gitirana Gomes Ferreira, and Paulo A. Cauchick Miguel. 2013. "Product-Service Systems: A Literature Review on Integrated Products and Services." *Journal of Cleaner Production* 47 (May):222–31. https://doi.org/10.1016/j.jclepro.2012.12.028.

Box, Don, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. 2000. "Simple Object Access Protocol (SOAP) 1.1."

Bray, Tim, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. 1997. "Extensible Markup Language (XML)." *World Wide Web Journal* 2 (4):27–66.

Bucchiarone, Antonio, Hernán Melgratti, and Francesco Severoni. 2007. "Testing Service Composition." In .

Camarinha-Matos, Luis M, and Walter Vieira. 1999. "Intelligent Mobile Agents in Elderly Care." *Robotics and Autonomous Systems* 27 (1–2):59–75.

Cândido, Gonçalo Moreira. 2013. "Service-Oriented Architecture for Device Lifecycle Support in Industrial Automation."

Cavalieri, Sergio, and Giuditta Pezzotta. 2012. "Product–Service Systems Engineering: State of the Art and Research Challenges." *Computers in Industry* 63 (4):278–288.

Chou, Timothy. 2010. *Introduction to Cloud Computing*. Cloudbook.

Christensen, Erik, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. 2001. "Web Services Description Language (WSDL) 1.1."

Christensen, James H. 1994. "Holonic Manufacturing Systems: Initial Architecture and Standards Directions." *Proc 1st Euro Wkshp on Holonic Manufacturing Systems*.

Colombo, Armando W, Thomas Bangemann, Statmatis Karnouskos, Jerker Delsing, Petr Stluka, Robert Harrison, Francois Jammes, and Jose L Lastra. 2014. "Industrial Cloud-Based Cyber-Physical Systems." *The IMC-AESOP Approach*.

Dassisti, Michele, Hervé Panetto, Angela Tursi, and Michele De Nicolo. 2008. "Ontology-Based Model for Production-Control Systems Interoperability." In , 527–43.

Davidsen, Leif. 2014. "IBM MQ: Integrated Messaging to Connect Your Enterprise (White Paper (External)-USEN)." 2014. http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&htmlfid=WSW14195USEN#loaded.

Di, Giovanni, Oliviu Matei, Sebastian Scholze, Dragan Stokic, José Barata, and Claudio Cenedese. 2016. "A Platform to Support the Product Servitization." *International Journal of Advanced Computer Science and Applications* 7 (2). https://doi.org/10.14569/IJACSA.2016.070254.

Di Orio, Giovanni. 2013. "Adapter Module for Self-Learning Production Systems." Faculdade de Ciências e Tecnologia. https://run.unl.pt/handle/10362/10402.

Dustdar, Schahram, and Wolfgang Schreiner. 2005. "A Survey on Web Services Composition." *International Journal of Web and Grid Services* 1 (1):1–30.

ElMaraghy, Hoda A. 2005. “Flexible and Reconfigurable Manufacturing Systems Paradigms.” *International Journal of Flexible Manufacturing Systems* 17 (4):261–76.

Erl, Thomas. 2005. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall. http://dspace.siu.ac.th/handle/1532/177.

Frei, Regina, Jose Barata, and Mauro Onori. 2007. “Evolvable Production Systems Context and Implications.” In , 3233–38. IEEE.

Goedkoop, Mark J., Cees JG Van Halen, H. Te Riele, Peter JM Rommens, and others. 1999. “Product Service Systems, Ecological and Economic Basics.” *Report for Dutch Ministries of Environment (VROM) and Economic Affairs (EZ)* 36 (1):1–122.

Goldman, Steven L. 1995. *Agile Competitors and Virtual Organizations: Strategies for Enriching the Customer*. Van Nostrand Reinhold Company.

Huhns, Michael N. 2002. “Agents as Web Services.” *IEEE Internet Computing* 6 (4):93–95.

Jammes, F., H. Smit, J.L. Martinez Lastra, and I.M. Delamer. 2005. “Orchestration of Service-Oriented Manufacturing Processes.” In , 1:617–24. IEEE. https://doi.org/10.1109/ETFA.2005.1612580.

Jassbi, Javad, Giovanni Di Orio, Diogo Barata, and José Barata. 2014. “The Impact of Cloud Manufacturing on Supply Chain Agility.” In *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, 495–500. IEEE. http://ieeexplore.ieee.org/abstract/document/6945563/.

Jordan, Diane, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, and Yaron Goland. 2007. “Web Services Business Process Execution Language Version 2.0.” *OASIS Standard* 11 (120):5.

Josuttis, Nicolai M. 2007. *SOA in Practice: The Art of Distributed System Design*. O’Reilly Media, Inc.

Juehling, Erik, Meike Torney, Christoph Herrmann, and Klaus Droeder. 2010. “Integration of Automotive Service and Technology Strategies.” *CIRP Journal of Manufacturing Science and Technology* 3 (2):98–106.

Kavantzas, Nickolaos. 2004. “Web Services Choreography Description Language (Ws-Cdf) Version 1.0.” *Http://www. w3. org/TR/ws-Cdl-10/*.

Koestler, Arthur. 1968. “The Ghost in the Machine.”

Komoda, Norihisa. 2006. “Service Oriented Architecture (SOA) in Industrial Systems.” In , 1–5. IEEE.

Koren, Y., M. Shpitalni, P. Gu, and S.J. Hu. 2015. “Product Design for Mass-Individualization.” *Procedia CIRP* 36:64–71. https://doi.org/10.1016/j.procir.2015.03.050.

Koren, Yoram. 2010. *The Global Manufacturing Revolution: Product-Process-Business Integration and Reconfigurable Systems*. Vol. 80. John Wiley & Sons.

Koren, Yoram, Uwe Heisel, Francesco Jovane, Toshimichi Moriwaki, Gumter Pritschow, Galip Ulsoy, and Hendrik Van Brussel. 1999. “Reconfigurable Manufacturing Systems.” *CIRP Annals-Manufacturing Technology* 48 (2):527–40.

Kumar, Ashok. 2007. “From Mass Customization to Mass Personalization: A Strategic Transformation.” *International Journal of Flexible Manufacturing Systems* 19 (4):533–47. https://doi.org/10.1007/s10696-008-9048-6.

Leitão, Paulo. 2004. “An Agile and Adaptive Holonic Architecture for Manufacturing Control.”

Licklider, Joseph CR. 1963. “Memorandum for Members and Affiliates of the Intergalactic Computer Network.” *M. A. A. Ot IC Network (Ed.). Washington DC: KurzweilAI. Ne*.

Manzini, Ezio, Carlo Vezzoli, and Garrette Clark. 2001. “Product Service Systems: Using an Existing Concept as a New Approach to Sustainability.” *Journal of Design Research* 1 (2):12–18.

Marinos, Alexandros, and Gerard Briscoe. 2009. “Community Cloud Computing.” In , 5931:472–84. Springer.

Mehrabi, Mostafa G, A Galip Ulsoy, and Yoram Koren. 2000. “Reconfigurable Manufacturing Systems: Key to Future Manufacturing.” *Journal of Intelligent Manufacturing* 11 (4):403–19.

Mell, Peter, and Tim Grance. 2009. “Perspectives on Cloud Computing and Standards.” *USA, NIST*.

Monostori, László, József Váncza, and Soundar RT Kumara. 2006. “Agent-Based Systems for Manufacturing.” *CIRP Annals-Manufacturing Technology* 55 (2):697–720.

Nagorny, Kevin, Armando Walter Colombo, and Uwe Schmidtmann. 2012. “A Service- and Multi-Agent-Oriented Manufacturing Automation Architecture.” *Computers in Industry* 63 (8):813–23. https://doi.org/10.1016/j.compind.2012.08.003.

Natis, Yefim V. 2003. “Service-Oriented Architecture Scenario.”

Neves, P, and J Barata. 2009. “Evolvable Production Systems.[in:] Assembly and Manufacturing, 2009. ISAM 2009.” In , 189.

Okino, Norio. 1993. “Bionic Manufacturing Systems.” In , 73–95.

Oliveira, José António Barata de. 2003. “Coalition Based Approach for Shop Floor Agility–a Multiagent Approach.” http://run.unl.pt/handle/10362/2483.

Onori, Mauro. 2002. “Evolvable Assembly Systems: A New Paradigm?” In .

Onori, Mauro, José Barata, and Regina Frei. 2006. “Evolvable Assembly Systems Basic Principles.” *Information Technology for Balanced Manufacturing Systems*, 317–28.

Papazoglou, Mike P., and Willem-Jan van den Heuvel. 2007. “Service Oriented Architectures: Approaches, Technologies and Research Issues.” *The VLDB Journal* 16 (3):389–415. https://doi.org/10.1007/s00778-007-0044-3.

Peltz, Chris. 2003. “Web Services Orchestration and Choreography.” *Computer* 36 (10):46–52.

Pine, B Joseph. 1993. *Mass Customization: The New Frontier in Business Competition*. Harvard Business Press.

Poizat, Pascal, and Gwen Salaün. 2012. “Checking the Realizability of BPMN 2.0 Choreographies.” In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, 1927–1934. SAC ’12. New York, NY, USA: ACM. https://doi.org/10.1145/2245276.2232095.

Pratl, Gerhard, Dietmar Dietrich, Gerhard P. Hancke, and Walter T. Penzhorn. 2007. “A New Model for Autonomous, Networked Control Systems.” *IEEE Transactions on Industrial Informatics* 3 (1):21–32. https://doi.org/10.1109/TII.2007.891308.

ProSEco Consortium. 2014. “D100.4 - Public_ProSEco_Concept.”

Qu, Min, Suihuai Yu, Dengkai Chen, Jianjie Chu, and Baozhen Tian. 2016. “State-of-the-Art of Design, Evaluation, and Operation Methodologies in Product Service Systems.” *Computers in Industry* 77:1–14.

Ren, Lei, Lin Zhang, Fei Tao, Chun Zhao, Xudong Chai, and Xinpei Zhao. 2015. “Cloud Manufacturing: From Concept to Practice.” *Enterprise Information Systems* 9 (2):186–209. https://doi.org/10.1080/17517575.2013.839055.

Ren, Lei, Lin Zhang, Lihui Wang, Fei Tao, and Xudong Chai. 2017. “Cloud Manufacturing: Key Characteristics and Applications.” *International Journal of Computer Integrated Manufacturing* 30 (6):501–15. https://doi.org/10.1080/0951192X.2014.902105.

Ribeiro, Luis, and Jose Barata. 2011. “Re-Thinking Diagnosis for Future Automation Systems: An Analysis of Current Diagnostic Practices and Their Applicability in Emerging IT Based Production Paradigms.” *Computers in Industry* 62 (7):639–59. https://doi.org/10.1016/j.compind.2011.03.001.

Ribeiro, Luís, José Barata, and Pedro Mendes. 2008. “MAS and SOA: Complementary Automation Paradigms.” *Innovation in Manufacturing Networks*, 259–268.

Rosen, Mike. 2008. “BPM and SOA: Orchestration or Choreography.” *BPTrends, Www. Bptrends. Com*.

Rubino, Sara Córdoba, Wimer Hazenberg, and Menno Huisman. 2011. *Meta Products: Meaningful Design for Our Connected World*. Bis Publishers.

Russell, Stuart J., and Peter Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Englewood Cliffs, N.J: Prentice Hall.

Sakao, Tomohiko, Gunilla Ölundh Sandström, and Detlef Matzen. 2009. “Framing Research for Service Orientation of Manufacturers through PSS Approaches.” *Journal of Manufacturing Technology Management* 20 (5):754–78.

Scholze, Sebastian, Ana Teresa Correia, and Dragan Stokic. 2016. “Novel Tools for Product-Service System Engineering.” *Procedia CIRP*, Product-Service Systems across Life Cycle, 47 (January):120–25. https://doi.org/10.1016/j.procir.2016.03.237.

Schumacher, Michael. 2001. *Objective Coordination in Multi-Agent System Engineering: Design and Implementation*. Berlin, Heidelberg: Springer-Verlag.

Sethi, Andrea Krasa, and Suresh Pal Sethi. 1990. “Flexibility in Manufacturing: A Survey.” *International Journal of Flexible Manufacturing Systems* 2 (4):289–328.

Shah, R. 2003. “Lean Manufacturing: Context, Practice Bundles, and Performance.” *Journal of Operations Management* 21 (2):129–49. https://doi.org/10.1016/S0272-6963(02)00108-0.

Sundin, Erik. 2009. “Life-Cycle Perspectives of Product/service-Systems: In Design Theory.” *Introduction to Product/service-System Design*, 31–49.

Tao, F., L. Zhang, V. C. Venkatesh, Y. Luo, and Y. Cheng. 2011. “Cloud Manufacturing: A Computing and Service-Oriented Manufacturing Model.” *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 225 (10):1969–76. https://doi.org/10.1177/0954405411405575.

Ueda, Kanji. 1992. “A Concept for Bionic Manufacturing Systems Based on DNA-Type Information.” In , 853–63. North-Holland Publishing Co.

Van Brussel, Hendrik, Jo Wyns, Paul Valckenaers, Luc Bongaerts, and Patrick Peeters. 1998. “Reference Architecture for Holonic Manufacturing Systems: PROSA.” *Computers in Industry* 37 (3):255–74.

Vandermerwe, Sandra, and Juan Rada. 1988. “Servitization of Business: Adding Value by Adding Services.” *European Management Journal* 6 (4):314–24. https://doi.org/10.1016/0263-2373(88)90033-3.

Vincent Wang, Xi, and Xun W. Xu. 2013. “An Interoperable Solution for Cloud Manufacturing.” *Robotics and Computer-Integrated Manufacturing* 29 (4):232–47. https://doi.org/10.1016/j.rcim.2013.01.005.

White, Stephen A. 2004. “Introduction to BPMN.” *IBM Cooperation* 2 (0):0.

Wooldridge, M. 2009. *An Introduction to MultiAgent Systems*. Wiley. https://books.google.pt/books?id=X3ZQ7yeDn2IC.

Xu, Xun. 2012. “From Cloud Computing to Cloud Manufacturing.” *Robotics and Computer-Integrated Manufacturing* 28 (1):75–86. https://doi.org/10.1016/j.rcim.2011.07.002.

Yu, Min, Weimin Zhang, and Horst Meier. 2008. “Modularization Based Design for Innovative Product-Related Industrial Service.” In , 1:48–53. IEEE.

Yusuf, Y.Y, M Sarhadi, and A Gunasekaran. 1999. “Agile Manufacturing:” *International Journal of Production Economics* 62 (1–2):33–43. https://doi.org/10.1016/S0925-5273(98)00219-9.

Annex I

PES Deployable Solution (in JSON format), sent from the Service Composition Engineering Tool into the PES Deployment platform:


```
{
 "hasConfigurations": [
    {
      "id": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-d6489d2964a4",
      "belongTo":
"pt.uninova.proseco.electrolux.tool.services.SpecificCollectorService",
      "type":
"pt.uninova.proseco.electrolux.tools.pes.ontology.SpecificCollectorConfiguration",
       "value": "{\"id\":\"CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
  d6489d2964a4\",\"belongTo\":\"pt.uninova.proseco.electrolux.tool.services.SpecificColle
  ctorService\",\"type\":\"pt.uninova.proseco.electrolux.tools.pes.ontology.SpecificColle
  ctorConfiguration\",\"outputModel\":{\"associated_PesId\":\"Uninova_Fridge_test_1\",\"s
  erviceType\":\"pt.uninova.proseco.electrolux.tool.services.SpecificCollectorService\",\
  "repositoryType\":\"pt.uninova.proseco.electrolux.tool.services.SpecificCollectorReposi
  toryService\",\"associated_ConfigId\":\"CollectorConfiguration_test93e4d4d8-0256-4c97-
  8edc-
  d6489d2964a4\",\"root\":{\"implementingClass\":\"pt.uninova.proseco.electrolux.model.Da
  taCollectionModel\",\"hasFields\":[]}}}"
    },
    {
      "id": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-644d7adc6bc9",
      "belongTo": "pt.uninova.proseco.electrolux.services.SpecificDataAnalyserService",
      "type":
"pt.uninova.proseco.electrolux.tools.pes.ontology.SpecificDataAnalyserConfiguration",
```

```
      "value":
"{\"AnalisysType\":\"mean_value\",\"id\":\"DataAnalyserConfiguration_test0e3f56da-62d2-
4d1f-985a-
644d7adc6bc9\",\"belongTo\":\"pt.uninova.proseco.electrolux.services.SpecificDataAnalyse
rService\",\"type\":\"pt.uninova.proseco.electrolux.tools.pes.ontology.SpecificDataAnaly
serConfiguration\",\"outputModel\":{\"associated_PesId\":\"Uninova_Fridge_test_1\",\"ser
viceType\":\"pt.uninova.proseco.electrolux.tool.services.SpecificCollectorService\",\"re
positoryType\":\"pt.uninova.proseco.electrolux.tool.services.SpecificCollectorRepository
Service\",\"associated_ConfigId\":\"CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4\",\"root\":{\"implementingClass\":\"pt.uninova.proseco.electrolux.model.Dat
aCollectionModel\",\"hasFields\":[]}}}"
      "bpmn": "..."
    },
    {
      "compElems": {
        "startEvents": {
          "StartEvent_1": {
            "outgoingFlowIds": [
              "SequenceFlow_17ftfhf"
            ],
            "id": "StartEvent_1"
          }
        },
        "endEvents": {
          "EndEvent_0tzsjpt": {
            "incomingFlowIDs": [
              "SequenceFlow_0f6lf9d"
            ],
            "id": "EndEvent_0tzsjpt"
          }
        },
        "serviceTasks": {
          "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-644d7adc6bc9": {
            "serviceName": "Specific Data Analyser",
            "incomingFlowIDs": [
              "SequenceFlow 1urxz6q"
            ],
            "outgoingFlowIds": [
              "SequenceFlow_0f6lf9d"
            ],
            "id": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-644d7adc6bc9"
          },
          "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-d6489d2964a4": {
            "serviceName": "Specific Collector",
            "incomingFlowIDs": [
              "SequenceFlow_17ftfhf"
            ],
            "outgoingFlowIds": [
              "SequenceFlow_1urxz6q"
            ],
            "id": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-d6489d2964a4"
          }
        },
        "annotations": {
          "TextAnnotation_17qtsmp": {
            "paramsText": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-
644d7adc6bc9\nSequenceFlow_1urxz6q:0:0:5:Periodic:0:0:10",
            "id": "TextAnnotation_17qtsmp"
          },
          "TextAnnotation_1gogydw": {
            "paramsText": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4",
            "id": "TextAnnotation_1gogydw"
          }
        },
        "flows": {
          "SequenceFlow_17ftfhf": {
            "SourceId": "StartEvent_1",
            "TargetId": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4",
            "id": "SequenceFlow_17ftfhf"
          },
          "SequenceFlow_0f6lf9d": {
            "SourceId": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-
644d7adc6bc9",
            "TargetId": "EndEvent_0tzsjpt",
```

```
            "id": "SequenceFlow_0f6lf9d"
          },
          "SequenceFlow_1urxz6q": {
            "SourceId": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4",
            "TargetId": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-
644d7adc6bc9",
            "id": "SequenceFlow_1urxz6q"
          }
        }
      },
      "runningServices": {
        "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-644d7adc6bc9": {
          "serviceID": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-
644d7adc6bc9",
          "annot": {
            "paramsText": "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-
644d7adc6bc9\nSequenceFlow_1urxz6q:0:0:5:Periodic:0:0:10",
            "id": "TextAnnotation_17qtsmp"
          },
          "isRestartable": false,
          "continuosOperationMode": false,
          "isStarter": false,
          "flowspecs": {
            "SequenceFlow_1urxz6q": {
              "flowId": "SequenceFlow_1urxz6q",
              "sourceId": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4",
              "sourceInfo": {},
              "flowtype": "Periodic",
              "period": 10,
              "hours": 0,
              "minutes": 0,
              "seconds": 5,
              "delayTimeFromSource": 5,
              "delayTimeOfPesStart": 5,
              "settled": true
            }
          },
          "dataOutputIds": [],
          "receivers": [],
          "delayFromStartSettled": true,
          "delayFromStart": 5
        },
        "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-d6489d2964a4": {
          "serviceID": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4",
          "annot": {
            "paramsText": "CollectorConfiguration_test93e4d4d8-0256-4c97-8edc-
d6489d2964a4",
            "id": "TextAnnotation_1gogydw"
          },
          "isRestartable": false,
          "continuosOperationMode": false,
          "isStarter": true,
          "flowspecs": {},
          "dataOutputIds": [
            "SequenceFlow_1urxz6q"
          ],
          "receivers": [
            "DataAnalyserConfiguration_test0e3f56da-62d2-4d1f-985a-644d7adc6bc9"
          ],
          "delayFromStartSettled": true,
          "delayFromStart": 0
        }
      },
      "id": "Comp_e2f0e2e6-fbf7-4538-bb4a-32aafa20a471",
      "type": "ServiceCompositionConfiguration"
    }
  ],
  "id": "Uninova_Fridge_test_1",
  "creator": "Creator not specified"
}
```