Ecosystem for COllaborative Manufacturing PrOceSses – Intra- and Interfactory Integration and AutomaTION
(Grant Agreement No 723145)

COMPOSITION Brokering and Matchmaking Components I

Date: 2018-04-30

Version 1.0

Published by the COMPOSITION Consortium

Dissemination Level: Public

Co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145

Document control page

Document file: D6.9 COMPOSITION Brokering and Matchmaking components I v1.0.docx
Document version: 1.0
Document owner: CNET

Work package: WP6
Task: T6.5
Deliverable type: OTHER

Document status: ☒ Approved by the document owner for internal review
☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Mathias Axling (CNET),	2018-03-23	Initial TOC, Section 4 - Architecture
0.2	Alexandros Nizamis, Vagia Rousopoulou (CERTH)	2018-04-03	Content to Section 5 – Related Works, Section 6 - Ontology
0.3	Alexandros Nizamis (CERTH)	2018-04-16	Section 6 Matchmaker Implementation
0.4	Alexandros Nizamis (CERTH)	2018-04-18	Section 6 Matchmaker Implementation update
0.5	Alexandros Nizamis, Dimosthenis Ioannidis (CERTH)	2018-04-23	Sections 7, 8, 9
0.6	Mathias Axling (CNET)	2018-04-24	Sections 1, 3
0.7	Mathias Axling (CNET)	2018-04-24	Edited acronym table
0.8	Mathias Axling (CNET)	2018-04-24	Ready for peer review
1.0		2018-04-27	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Christian Beecks (FIT-UC²)	2018-04-26	
Vasiliki Charisi (ATL)	2018-04-25	

Index:

1 Executive Summary

This report describes the first results of Task 6.5 Brokering and Matchmaking for Efficient Management of Manufacturing Processes. The Matchmaker is a core component of the COMPOSITION Collaborative Ecosystem, providing matching of the capabilities of buyers and sellers in the supply chain as well as ranking of offers during marketplace agent negotiations.

To this end, both syntactic and semantic matching of manufacturing capabilities is applied to find the best possible supplier to fulfil a request for a service, raw materials or products involved in the supply chain.

For measuring the similarity among offers and requests, well-established weighted similarity algorithms and metrics will be used and will be further extended if needed, in order to address the objective of COMPOSITION at the best possible way. Different decision criteria for supplier selection according to several qualitative and quantitative factors will be considered (e.g. size of buyer's organization, cost, time, distance, due date, quality, price, technical capability, financial position, past performance, attitude, flexibility, etc.). Special focus will be given in dealing with the trade-off between performance and quality of matching, in order to provide responses in a reasonable time while at the same time minimization of computational complexities will be targeted.

This report describes the work that has been done from M5 (Task 6.5 starts) to M20 (date of this deliverable). The final results of Task 6.5 and all the updates of the Matchmaker from M20 until M34 will be reported in the report D6.10 COMPOSITIOWPN Brokering and Matchmaking components II.

2 Abbreviations and Acronyms

Acronym	Meaning
API	Application Programming Interface
CXL	COMPOSITION eXchange Language
FITMAN	Future Internet Technologies for MANufacturing industries
FITMAN-SeMaSE	Metadata and Ontologies Semantic Matching Specific Enabler
GRDDL	Gleaning Resource Descriptions from Dialects of Languages
IMPACT	Interactive Maryland Platform for Agents Collaborating Together
JSON	JavaScript Object Notation
LARKS	Language for Advertisement and Request for Knowledge Sharing
MASON	Manufacturing's Semantics Ontology
MSDL	Manufacturing Service Description Language
OWL	Web Ontology Language
RETSINA	Reusable Task Structured-based Intelligent Network Agents
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	Simple Protocol and RDF Query Language
WP	Work Package

3 Introduction

3.1 Purpose, context and scope of this deliverable

This deliverable presents the work carried out and the first results of the Task 6.5 Brokering and Matchmaking for Efficient Management of Manufacturing Processes. The work has been carried out in Work Package 6 (WP6), “COMPOSITION Collaborative Ecosystem”. The task is tightly integrated with Task 6.4 “Collaborative manufacturing services ontology and language”, the results of which have been described in D6.7 Collaborative manufacturing services ontology and language I. This report will include an overview of the integration with the manufacturing services ontology.

This deliverable will be followed by D6.10 “COMPOSITION Brokering and Matchmaking components II”, which will provide an updated description at M34 of the project.

3.2 Content and structure of this deliverable

The report will provide an overview of the role of the Matchmaker component in the COMPOSITION system, together with a description of the design and interfaces of the Matchmaker and its dependencies on other components, specifically the Collaborative Manufacturing Services Ontology. Furthermore, planned future work and lessons learned will be reported. The document is structured as follows:

Section 4 describes how the Matchmaker component is integrated in the overall COMPOSITION architecture and describes its interactions with and dependencies on other COMPOSITION components. Special attention is given to interactions with the Marketplace agents and the Collaborative Manufacturing Services Ontology.

Section 5 includes a brief description of state-of-the-art analysis and related works presentation performed for the Matchmaker.

Section 6 provides a detailed description of the design of the Matchmaker with emphasis at the semantic rules and the interconnections to the Collaborative Manufacturing Services Ontology.

Section 7 documents the first version of the Matchmaker Agent API and Matchmaker Offer API, that is used by the COMPOSITION Marketplace agents.

Section 8 outlines the next steps of Task 6.5, which will be presented in deliverable D6.10 COMPOSITION Matchmaking and brokering components II at M34 when the task ends.

Section 9 is the conclusions section which provides a summary of contents of the deliverable and lessons learned.

4 Role of Brokering and Matchmaking components in COMPOSITION Architecture

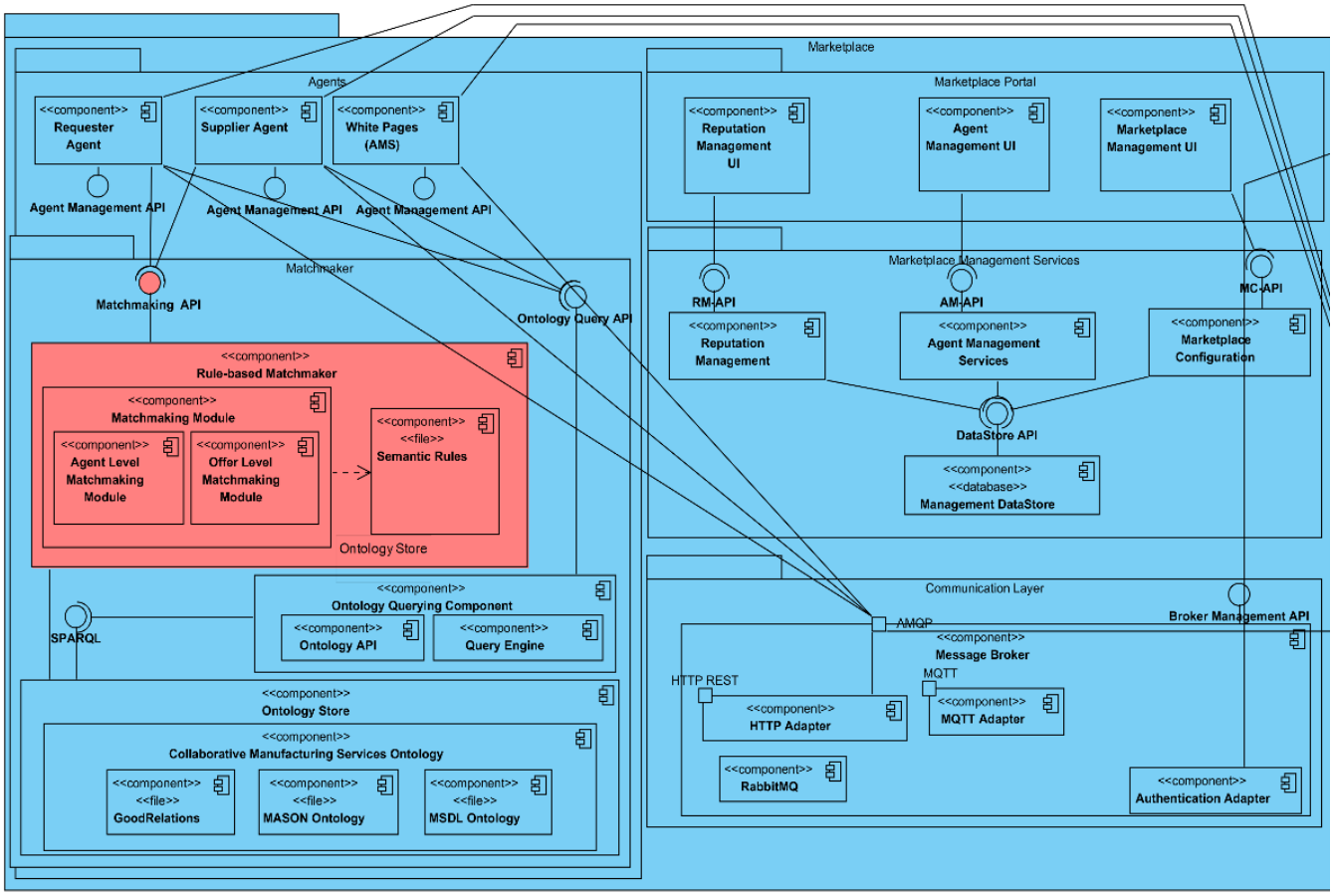

Figure 1: Matchmaker component in relation to COMPOSITION Collaborative Ecosystem architecture

As shown in Figure 1, the Brokering and Matchmaking components of the Rule-based Matchmaker (in red) are part of the Matchmaker package, which also includes the Ontology Querying Component and Ontology Store. The Matchmaker package is in turn part of the Agents package.

The Marketplace's agents use the Matchmaking API to get selections of suitable suppliers for call for proposal (CFP) to be sent by a requester agent and to evaluate the offers sent by supplier agents in response to the CFP. The Ontology Query Component provides management and querying of the Collaborative Manufacturing Services Ontology, via the exposed Ontology Query API interface. The Agents can update and query the Collaborative Manufacturing Services Ontology through the interface. The Rule-based Matchmaker component is connected directly to the Ontology Store on which it will apply rules in order to infer new knowledge from the Collaborative Manufacturing Services Ontology. The rules can be applied directly at the file system which contains the Ontology Store, or they can be applied to an Ontology Model which has been loaded in the memory. The design of the Matchmaker is reported in section 10 Design of Brokering and Matchmaking components. A detailed description of the Matchmaker APIs, interaction with Agents and Matchmaker deployment is provided in section 7 Matchmaker APIs and Deployment.

The Matchmaker is involved in Use Cases *UC-KLE-4 Scrap metal collection and bidding process*, *UC-KLE-7 Ordering raw materials*, *UC-ELDIA-1 Fill-level Notification – Contractual solid recyclable waste management*, *UC-ELDIA-2 Fill-level Notification – Contractual wood waste management*, *UC-ATL-3 Searching for recommended solutions, UC-ATL-1 Selling software/consultancy*, *UC-ATL-2 Searching for solutions* and *UC-ATL-3 Searching for recommended solutions*. These are described in D2.1 Industrial use cases for an Integrated Information Management System.

5 Related Works

There are several existing approaches related to manufacturing semantic representation and brokering, and matchmaking techniques. However, the related research mainly presents frameworks which are not completely related to manufacturing domain in connection with the supply chain domain. Furthermore, they are not exclusively designed for one system and they are not easily extended and adoptive by other agent-based ecosystems. The following related works are presented by the perspective of semantic representation and matchmaking.

LARKS

LARKS (Language for Advertisement and Request for Knowledge Sharing) (Sycara, 1999) based matchmaking engine was used in RETSINA 1 (Reusable Task Structured-based Intelligent Network Agents) infrastructure. It was a multi-agent infrastructure that was developed by the Carnegie Mellon University in Pittsburgh, USA and contained a matchmaking engine that relies on service matching. The matchmaking was based in LARKS which express advertisements and requests using the same language. Five different filters were contained in the aforementioned matchmaking engine: key-word-based matching, similarity matching, profile comparison matching, constraint matching and rule-based signature matching. Nevertheless, the RETSINA/LARKS matchmaking framework lacks of features matching. The used language is not focused on manufacturing domain and the LARKS matchmaker needs a manufacturing domain ontology which should be compatible with LARKS in order to be used as the content. Only then it is able to perform matching. However, due to the general nature of RETSINA/LARKS matchmaking engine, it is unable to capitalize on the advantages of the representation of the manufacturing specific services, tools and resources in order to be used in modern collaborative manufacturing ecosystems.

InfoSleuth

An agent-based system which performs different level information management activities was developed by MCC Inc., Texas, USA. This was InfoSleuth (Nodine, 2000). In the set of various agents which were offered by InfoSleuth, some Broker agents existed. These agents provide syntactic and semantic matchmaking between services' providers and requesters. In order to describe requests and advertisements a specific "InfoSleuth ontology" was used by the agents. The broker agents use textual comparisons for syntactic matchmaking of advertisements and queries. In the case of semantic matchmaking, broker agents apply SQL queries and then constraint matchmaking to queries' output in order to eliminate useless results based on advertisement capabilities and formal descriptions of the requests. However, the "InfoSleuth ontology" is not able to represent manufacturing services and resources as it is focused on advertisements and requests description. Thus, the broker agent's matchmaking engine is unable to perform a matchmaking process which covers the requirements of manufacturing collaborative ecosystems.

IMPACT

IMPACT (Interactive Maryland Platform for Agents Collaborating Together) (IMPACT, 2018) is an international research project led by the University of Maryland. It is related to software implementation that facilitates the creation, deployment, interaction, and collaborative aspects of software agents in a heterogeneous, distributed environment. IMPACT provides algorithms supporting a variety of applications including supply chain, logistics, and e-commerce. It supports multi-agent interactions and agent interoperability in an application independent manner. It provides a yellow pages server that performs basic matchmaking among agents based on weighted hierarchies. It maintains a verb and a noun hierarchy of synonyms and retrieval algorithms to compute similarities between given service specifications. So the IMPACT matchmaker uses only similarity and distance algorithms in order to perform matching. Moreover, the IMPACT matchmaker is not designed to support manufacturing domain concepts.

Digital Manufacturing Market

Digital Manufacturing Market (Ameri, 2012) is a multi-agent web-based framework that contains a manufacturing services ontology and a matchmaking mechanism which match a consumer's requirements with suppliers' manufacturing capabilities. The ontology used in this multi-agent framework is MSDL (Ameri, 2006), which stands for Manufacturing Service Description Language. MSDL is a manufacturing domain ontology which enables the representation of services and resources by describing manufacturing capabilities in four levels of abstraction: supply and demand level, shop-floor level, process level and machine level. Both advertisements and requests are expressed by agents using the MSDL as a common language. A middle agent, in order to find possible suppliers for a requested process, performs both features-based and taxonomy-based matchmaking. A list with possible suppliers is returned to the requester agent. The Digital Manufacturing

Market approach is the closest one with the presented matchmaker as it uses a common manufacturing ontology and performs semantic matching based on the services descriptions and terms related to this ontology. Besides some similarities in matchmaking logic for service and agent level matchmaking which will be presented in this report, the Digital Manufacturing Market solution does not use e-commerce concepts to extend the matchmaking process in an offer level in which the evaluation of the matching offers can be executed based in different qualitative and quantitative criteria.

FITMAN-SeMa

FITMAN-SeMa (Metadata and Ontologies Semantic Matching SE) (FITMAN-SeMa, 2018) is a component of FIWARE (FIWARE, 2018) for Industry 3 aims to solve interoperability problems in the collaboration of business processes. Furthermore, FITMAN-SeMa provides storing and retrieving functionalities for ontologies and triplets. By using various algorithms FITMAN-SeMa performs effective semantic matching. The FITMAN-SeMa is installable software which matches concepts between two different ontologies. This different approach may enable collaboration and possible matching of two different sources. Nevertheless, it is not a manufacturing agent-based eco-system dedicated solution. In order to achieve a higher level of interoperability FITMAN-SeMa introduces a solution which is not based in a central ontology. But this last feature makes the SeMa unable to extract conclusions from manufacturing domain in order to perform an efficient matchmaking of agents and services as it is not designed for this domain.

In conclusion of the related works analysis, it is perceived that most of the existing solutions are not exclusively designed for the manufacturing domain and lacks the necessary concepts that will enable efficient reasoning in term of manufacturing. Besides this, other approaches are completely related to this domain and lacks the ability to represent e-commerce means which are important for the reasoning and matchmaking over on-line marketplaces.

6 Design of Brokering and Matchmaking components

COMPOSITION Matchmaker is designed to be the core component of the COMPOSITION Broker. It supports semantic matching in terms of manufacturing capabilities, in order to find the best possible supplier to fulfil a request for a service or products involved in the supply chain. Different decision criteria for supplier selection, according to several qualitative and quantitative factors, are considered by the Matchmaker. Furthermore, the Matchmaker acts as a broker for the Marketplace's bidding processes and enables the automation of these processes as well. The Matchmaker evaluates the available offers from the providers in order to suggest the best one to the supplier.

In this chapter a brief analysis of Collaborative Manufacturing Services Ontology and Language is presented. The COMPOSITION Matchmaker's functionalities depend exclusively on the Collaborative Manufacturing Services Ontology and Language. The Matchmaker is designed to infer new knowledge by applying rules in terms of this ontology. Furthermore, since the Matchmaker component is built upon the Apache Jena API, the basic components of this API are presented in this chapter as well. Before the design and implementation details of the Matchmaker, the corresponding requirements are also presented.

6.1 Collaborative Manufacturing Services Ontology and Language

Collaborative Manufacturing Services Ontology is the knowledge base for the COMPOSITION Marketplace. It is used as a common vocabulary which offers interoperability and representation of both meanings and data. The Collaborative Manufacturing Services Ontology enables:

- The description of supply and demand entities participate in the Collaborative Ecosystem
- The description of manufacturing services, capabilities and resources for entities participate in the Collaborative Ecosystem

The Ecosystem agents will be able to make transactions as the above information will be described using this common ontology. For example an agent who requests a service or a product will be able to find a matching agent who supports this service or product based on knowledge base's information.

Figure 2below presents the main classes of the Collaborative Manufacturing Services Ontology and Language:

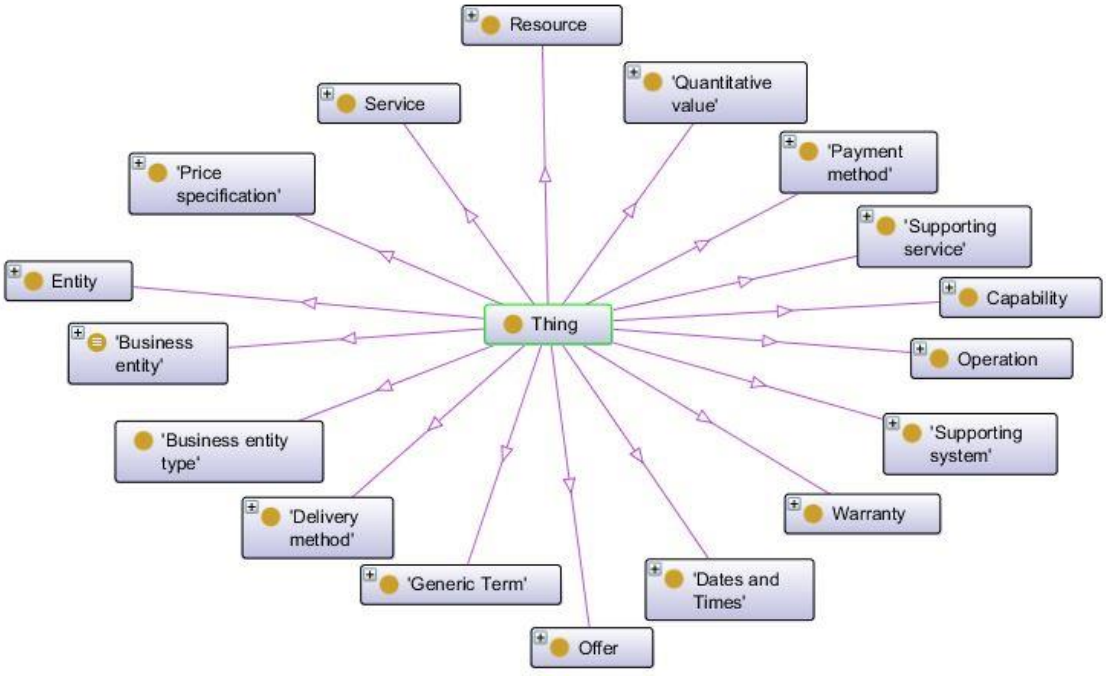

Figure 2: Collaborative Manufacturing Services Ontology Class Overview

The manufacturing domain should be supported as the Collaborative Manufacturing Services Ontology should be able to represent manufacturing services and resources. For this reason, MSDL (Ameri, 2006) and MASON (Lemaignan, 2006) ontologies are imported to the COMPOSITION Ontology as they are manufacturing domain

specific and they offer a large variety of classes and properties about this domain. Furthermore, the COMPOSITION Marketplace should be able to support collaboration mechanism between business entities. It should be able to describe relations and transactions between supply and demand entities which participate in the Marketplace. This need leads us to import the GoodRelations Language (GoodRelations Language, 2018) ontology which is one of the most well-known and widely used ontologies in e-commerce domain. All the aforementioned ontological resources were imported and re-engineered using Neon Methodology (M. C. Suárez-Figueroa, 2010) in order to create a stable and consistent version of the Collaborative Manufacturing Services Ontology. The implemented ontology's classes which are depicted in the previous Figure 2are presented in more details in the following table:

Table 1: Collaborative Manufacturing Services Ontology Classes

Class name	Description
Business entity	Represents an Ecosystem Agent who has a service (e.g. manufacturing service) and provides or seeks an offer
Business entity type	Represents the legal form, the size and the position of a business entity in value chain
Service	Conceptualizes all operations and processes related to a product in an abstract level
Operation	Represents the processes of a service
Resource	Represents the total set of linked resources of a business entity
Supporting service	Represent services which are not basic services but are related to the basic one and support them
Supporting system	Represents some systems which support a business entity's services
Offer	Represents a public announcement of a business entity that provides or seeks a certain service or product
Warranty	Represents the duration and the scope of free services that will be provided to a customer in case of a possible malfunction or problem
Quantitative value	Represent the range of a certain property
Generic Term	Define common operations, materials and tools
Delivery method	Define the available delivery options for a service or product
Dates and Times	The days that a business entity has opening hours. Also represents the day of delivery or the day of availability of a service
Capability	Represents the capability of a service
Entity	Represents an entity as a result of a manufacturing process and describe its geometric flaw and entity, assembly entity and raw material
Price specification	Specifies the price of a unit, additional delivery costs and additional costs related to a payment method
Payment method	Describes the available procedures for transferring the requested amount for a purchase

6.2 Apache Jena API

Apache Jena (Apache Jena, 2018) is a free and open source Java framework for building Semantic Web and Linked Data applications. The main component of this framework is an API that provides data extraction from RDF graphs as well as writing to them. The graphs are defined as an abstract model. A model can collect data from files, databases, URLs or a combination of these. Jena provides a programmatic environment for

RDF, RDFS and OWL, SPARQL, GRDDL, and includes a rule-based inference engine. Figure 3 below represents Jena framework's architecture.

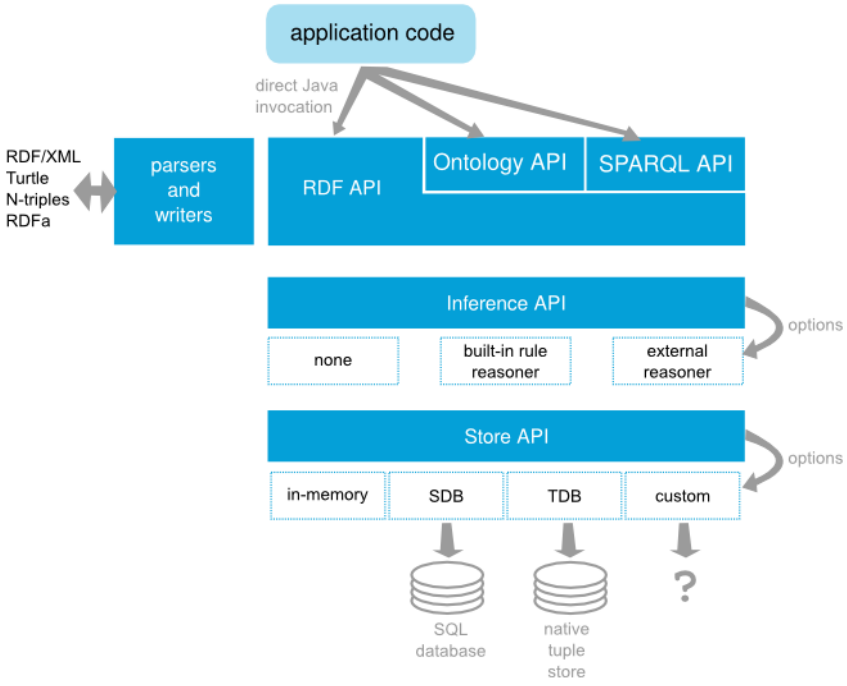

Figure 3: Apache Jena's framework architecture (Apache Jena, 2018)

RDF API

RDF can be better comprehended if it is represented in the form of node and arc diagrams, namely in RDF graphs. Each relationship points only to one direction. Part of the RDF graphs is resources. A resource is some entity. It could be a web resource or it could be a concrete physical thing. It could also be an abstract idea. Resources are named by a Uniform Resource Identifier (URI).

Jena is a Java API which can be used to create and manipulate RDF graphs. The interfaces representing resources, properties and literals are called Resource, Property and Literal respectively. In Jena, a graph is called a model and is represented by the Model interface.

The basic concepts of RDF containers in Jena are the following three:

- graph, a mathematical view of the directed relations between nodes in a connected structure
- Model, a rich Java API with many convenience methods for Java application developers
- Graph, a simpler Java API intended for extending Jena's functionality.

Ontology API

Jena allows a programmer to specify, in an open, meaningful way the concepts and relationships that collectively characterize some domain. The advantage of ontology is that it is an explicit, first-class description; it can be published and reused for different purposes.

There is a multitude of different ontology languages available for modelling ontology information on the semantic web. They range from the most expressive, OWL to the weakest, RDFS. Jena Ontology API aims to provide a coherent programming interface for ontology application development. The Ontology API is independent of the language used: the Java class names are not specific to the underlying language.

In order the distinction between various representations to be clear, each of the ontology languages has a profile, which lists the permitted constructs and the names of the classes and properties. The profile is bound to an ontology model, which is an extended version of Jena's Model class. The base Model allows access to the statements in a collection of RDF data. Jena ontology interface provides support for the kinds of constructs expected to be in ontology: classes (in a class hierarchy), properties (in a property hierarchy) and individuals.

SPARQL API

SPARQL is a query language and a protocol for accessing RDF designed. As a query language, SPARQL is "data-oriented" in that it only queries the information held in the models and does not infer in the query language itself. Jena model creates triples on-demand in order to give the impression that they already exist, including OWL reasoning. SPARQL takes the description of the application demands, in the form of a query, and returns that information, in the form of a set of bindings or an RDF graph.

Interference API

The Jena inference subsystem is designed to allow a range of inference engines or reasoners to be plugged into Jena. Such engines are used to derive additional RDF assertions which are entailed from some base RDF together with any optional ontology information and the axioms and rules associated with the reasoner.

Store API

Two individual parts of the Store API are TDB and SDB, as shown in Figure 3.

TDB is a component of Jena for RDF storage and query. It is a fast persistent triple store that stores directly to disk and supports the full range of Jena APIs. TDB can be used as a high performance RDF store on a single machine. A TDB store can be accessed and managed with the provided command line scripts and via the Jena API. When accessed using transactions, a TDB dataset is protected against corruption, unexpected process terminations and system crashes. On the other side, SDB uses an SQL database for the storage and query of RDF data. Many databases are supported, both Open Source and proprietary. An SDB store can be accessed and managed with the provided command line scripts and via the Jena API.

6.3 Matchmaker Requirements

The design and the implementation of the COMPOSITION Matchmaker were driven by the project's requirements. The main requirements related to the matchmaking component are listed below:

Table 2: Main Matchmaker Requirements

Requirement Number	Title	Short Description
COM-61	Suppliers' product/services shall be matched with a potential customers' needs/problems	This requirement relates to unite both suppliers and potential new customers in an automatic ecosystem, precisely matching the customers' needs with the companies' products and services. The system suggests for example a top five of potential suppliers, based on certain criteria, set by the customer
COM-86	The Matchmaker shall apply both syntactic and semantic matching	The Matchmaker shall apply both syntactic and semantic matching (both taxonomy-based and feature-based) in terms of manufacturing capabilities, in order to find the best possible supplier to fulfil a request for a service, raw materials or products involved in the supply chain
COM-87	Different similarity algorithms and metrics shall be supported by the Matchmaker	For measuring the similarity among offers and requests, well-established weighted similarity algorithms and metrics will be supported by the Matchmaker and will be further extended if needed, in order to address the objective of COMPOSITION at the best possible way
COM-88	Different decision criteria for supplier selection are supported by the Matchmaker	Different decision criteria for supplier selection according to several qualitative and quantitative factors shall be considered (e.g. size of buyer's organization, cost, time, distance, due date, quality, price, technical

		capability, financial position, past performance, attitude, flexibility, etc.) in matchmaking
COM-89	Matchmaker shall return a result within a reasonable time frame	The Matchmaker should respond within a reasonable time frame (e.g. 5 seconds)
COM-90	Ecosystem components should be deployed as Docker images	Docker gives ease of deployment and simpler integration of heterogeneous components. Exact configuration of target platform can be performed by the partner developing the component and setup is easy for other partners. Many third-party components are also available as Docker images
COM-148	Matchmaker and Agents components should be able to access and manipulate Marketplace Ontology	The matchmaker and the agent components should be able to access the Ontology Store. Based on type of agents, the should be able to infer knowledge or store and retrieve data from Collaborative Manufacturing Services Ontology

6.4 Rule-based Matchmaker Implementation Details

The COMPOSITION Semantic Matchmaker is built upon Apache Jena framework. The Semantic Matchmaker aims to infer new knowledge from the Collaborative Manufacturing Services Ontology based on semantic rules in order to perform matchmaking. In the overall COMPOSITION architecture the Matchmaker block contains the complete semantic framework of the project. This framework contains:

- Collaborative Manufacturing Services Ontology which initialize the Ontology Store (RDF triple store)
- Ontology Query Engine and the corresponding Ontology API which enable the manipulation of the Ontology Store by the Marketplace agents
- Rule-based Matchmaker which applies sets of semantic rules at the Ontology Store

The third of the aforementioned components will be analysed in this report as this one is about Brokering and Matchmaking. The other two components were presented at their corresponding report, D6.7 Collaborative manufacturing services ontology and language I (M14).

6.4.1 Semantic Rules

The Rule-based Matchmaking component contains sets of semantic rules. The semantic rules are commonly specified by means of an ontology language. These rules are used to infer new knowledge based on the existing one in the knowledge base/ontology and can be added as RDF triples. The rules are fired by reasoners which can be used and activated in applications. A reasoner is software able to infer logical consequences from a set of asserted facts or axioms. In the case of the Rule-based Matchmaker a rule-based reasoner offered by Jena API will be used. A rule for the rule-based reasoner is defined by a Java Rule object with a list of body terms (premises), a list of head terms (conclusions) and an optional name and optional direction. A term is a triple pattern, or an extended triple pattern or a call to a built-in primitive. A rule set is simply a List of Rules. The following image presents the simplified text rule syntax:

```
Rule      :=   bare-rule .
           or   [ bare-rule ]
         or   [ ruleName : bare-rule ]

bare-rule :=   term, ... term -> hterm, ... hterm    // forward rule
           or   bhterm <- term, ... term    // backward rule

hterm     :=   term
           or   [ bare-rule ]

term      :=   (node, node, node)           // triple pattern
           or   (node, node, functor)        // extended triple pattern
           or   builtin(node, ... node)      // invoke procedural primitive

bhterm      :=   (node, node, node)           // triple pattern

functor   :=   functorName(node, ... node)  // structured literal

node      :=   uri-ref                   // e.g. http://foo.com/eg
           or   prefix:localname          // e.g. rdf:type
           or   <uri-ref>          // e.g. <myscheme:myuri>
           or   ?varname                    // variable
           or   'a literal'                 // a plain string literal
           or   'lex'^^typeURI              // a typed literal, xsd:* type names supported
           or   number                      // e.g. 42 or 25.5
```

Figure 4: Jena Rules Syntax (Apache Jena, 2018)

A rule file has the main basic components:

- *@prefix* defines a prefix which can be used in the rules. The prefix is local to the rule file Example: @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
- *//* are comment lines
- *Triple patterns* – like a triple, but with some named variables instead of fixed parts
- *Rule "Body"* – Set of triple patterns, all of which must match.
- *Rule "Head"* – Set of triple patterns that will be asserted, when the body matches

Table 3: Jena Rule Example

Textual Format	Jena Rule Format
Business Entity X requests an offer And Business Entity X matches with Business Entity Y Which offers an Offer Y Then request X matches with Offer Y	@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. [exampleRule: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?x comp:matchesWith ?y) (?y v1:offers ?Offery) ->(?Offerx comp:matchingOffer ?Offery) //inferred knowledge is that offer x matches with offer y]

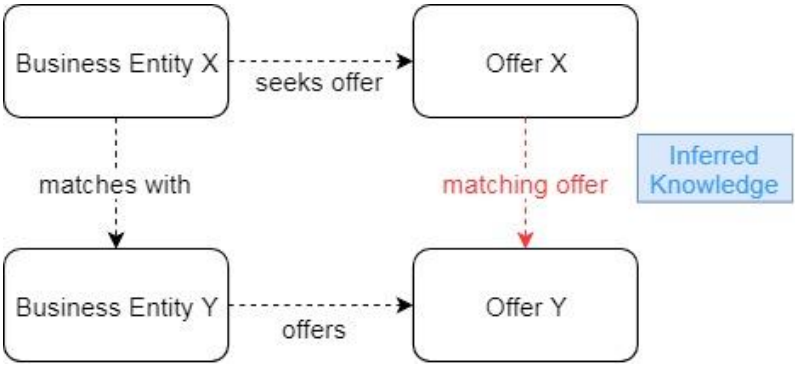

Figure 5: Jena Rule Example Representation

The above simplified Jena rule example explains how new knowledge can be inferred. A request (Offer X) can be matched to an offer (Offer Y) by this simple rule. The two instances, Offer X and Offer Y are connected with the object property 'matching offer'. This is the new knowledge that originally does not exist in the Ontology.

Furthermore, the Jena API offers a wide set of built-in primitives that can be included and used in rules files. The procedural primitives which can be called by the rules are each implemented by a Java object stored in a registry. Each primitive can be used in the rule body, the rule head or both. Some interest built-in primitives which many of them are used by the COMPOSITION Matchmaker are listed below. Moreover, additional/custom primitives can be created.

Table 4: Examples of Built-in Primitives

Built-in Primitive	Short Description
equal(?x,?y) notEqual(?x,?y)	Test if x=y (or x!= y). The equality test is semantic equality
lessThan(?x, ?y), greaterThan(?x, ?y) le(?x, ?y), ge(?x, ?y)	Test if x is <, >, <= or >= y
sum(?a, ?b, ?c) addOne(?a, ?c) min(?a, ?b, ?c) max(?a, ?b, ?c)	Sets c to be (a+b), (a+1), min(a,b), max(a,b)
remove(n, ...) drop(n, ...)	Remove the statement (triple) which caused the n^{th} body term of this rule to match. Drop will silently remove the triple(s) from the graph but not fire any rules as a consequence.
print(?x, ...)	Print a representation of each argument.
noValue(?x, ?p)	True if there is no known triple (x, p,)

6.4.2 Matchmaking Module

The Matchmaking Module is developed in Java and it is built upon the Apache Jena API. The Matchmaker is offered to other components through RESTful web services. Its core functionality is to receive Marketplace Agents' requests via Matchmaker API and to apply sets of semantic rules to the Ontology Store based on these requests. New knowledge will be inferred by the rules' appliance, and then the Matchmaking Module responses to the Agents by using the Matchmaker API. The next steps are followed by the Matchmaking Module:

1. The module receives requests by agent (requests are based on REST and HTTP)
2. The module accesses the Collaborative Manufacturing Services from the Ontology Store. An Ontology Model can be created in the memory or it can be accessed directly from the file system.
3. The module transforms the request from agents' COMPOSITION eXchange Language (CXL is in JSON format) compatible format to terms of the ontology and creates instances (if needed)

4. The module reads the Jena rules as a List of Jena rules files
5. A reasoner is selected. A reasoner can be created by calling an instance of a reasoner class or by retrieving from reasoner registry which contains instances indexed by URI assigned to the reasoner. The GenericRuleReasoner class is selected for the COMPOSITION Matchmaker purposes as it is a reasoner interface that is able to invoke any of the useful rule engine combinations.
6. The rules' list is set after the reasoner instance is created. This action indicates to the reasoner the set of rules that should execute
7. An inference model will be created after applying the reasoner to data.
8. The module accesses the information stored in inference model. The content of the inference model is the generated output after performing inference
9. The module transforms the inferred information to agents' CXL
10. The output is returned as a response via Matchmaker API (REST and HTTP) to the Agent in a format compatible to CXL

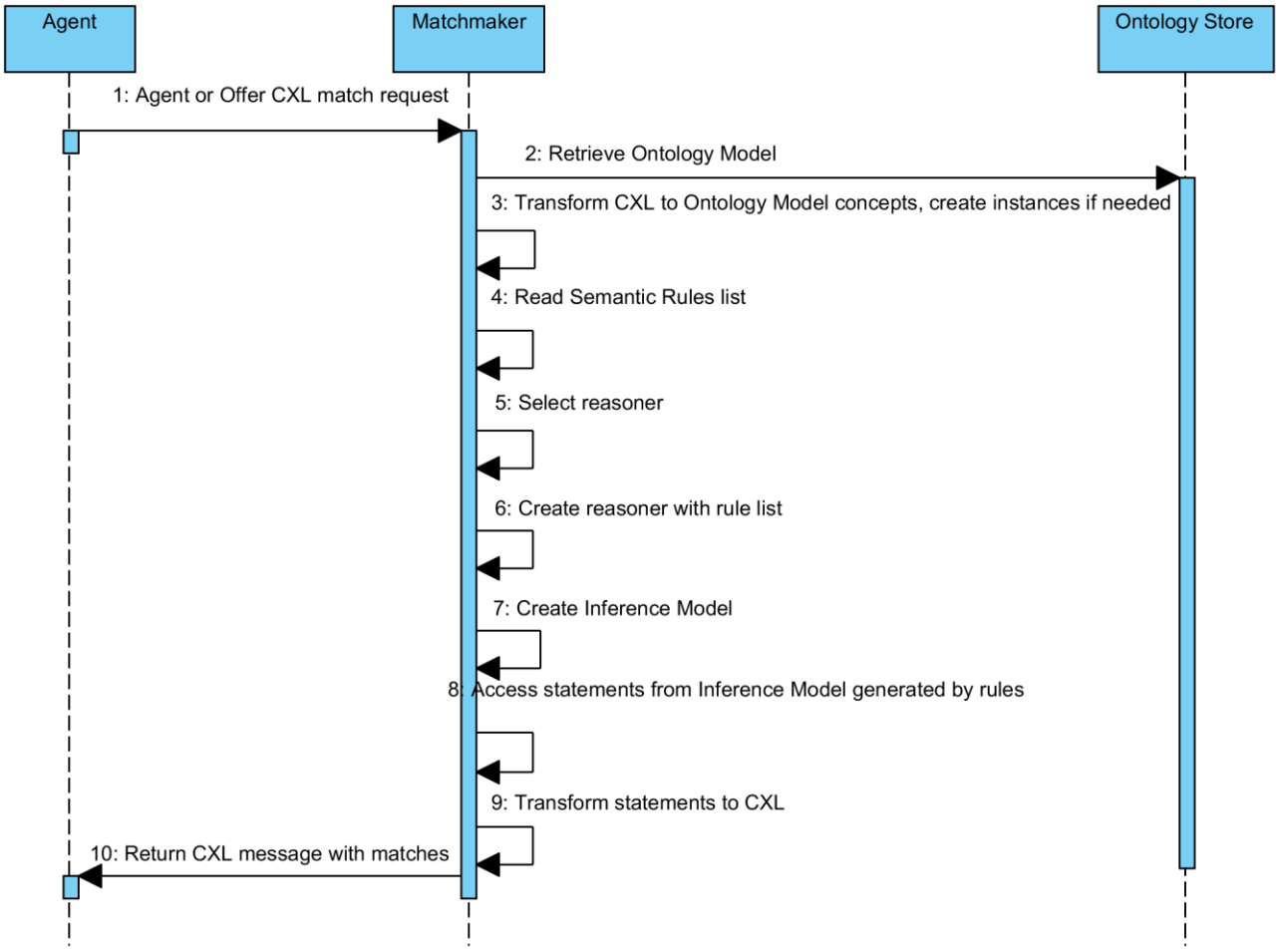

Figure 6: Agent to Matchmaker request sequence diagram

The Matchmaking Module contains two sub-modules. The Agent Level and the Offer Level matchmaking modules:

Agent Level Matchmaking

The Agent Level module aims to match Marketplace agents which are possible customers and suppliers. In this level of matching the Matchmaker applies rules which are based on ontology's classes: Business Entity, Generic Term, Capability, Service, Operation and Resource. The applied rules targets to infer knowledge that

enables the beginning of negotiation among the Marketplace stakeholders. The matchmaker indicates to a requester agent a list of possible supplier agents based on some requested criteria.

At this level of matching the semantic rules are focused on service level. For an agent who requests a service in the COMPOSITION Ecosystem, the Matchmaker will provide the agents which offers this service. In order to find possible providers of this service, the Matchmaker applies the following semantic rule based on terms of the ontology:

Table 5: Rule for Matching Business Entities

Textual Format	Jena Rule Format

Business Entity Y
requests an offer which
includes a service which
supports a specific operation Y
Business Entity X
offers a service which
supports an operation which
based on Generic Terms Catalog is
mapped with operation Y
Then
Entity Y matches with Entity X
	`@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.`
	`@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.`
	`@prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>.`
	`@prefix v1: <http://purl.org/goodrelations/v1#>.`
	`@prefix p1: <http://www.owl-ontologies.com/mason.owl#>.`
	`@prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>.`
	
	`[matchBusinessEntities:`
	
	`(?y rdf:type v1:BusinessEntity)`
	`(?y v1:seeksOffer ?Offery)`
	`(?Offery v1:includes ?Servicey)`
	`(?Servicey comp:seeksOperation ?Operationy)`
	`(?x rdf:type v1:BusinessEntity)`
	`(?x MSDL:hasService ?Servicex)`
	`(?Servicex comp:hasOperation ?Operationx)`
	`(?Operationx comp:mappedToCommonTerm ?Operationy)`
	`->`
	`(?y comp:matchesWith ?x)`
	
	`]`

Using the previous rule, the semantic matchmaker is able to match services, more precisely operations based on some common terms instances that exist in the Collaborative Manufacturing Services Ontology. Every business entity use its own terms to describe one of its offered services. But every one of these vendor specific terms will be mapped with a common generic term. In this way, on the one hand every business entity will be able to participate in the Marketplace and advertise its services, products etc. with its own terms. On the other hand, the Matchmaker will be able to match similar concepts in order to set the Marketplace capable to relate offers and requests among stakeholders or to find possible solutions for some Marketplace participants.

Moreover, the rules are extended in order to give a matchmaking result based also on some criteria by the requesters. For example, the requester can ask for a supplier who offers a specific service and has a Marketplace rating greater than a requested value. The following rule highlights the addition to the previous rule in order to offer the aforementioned capability.

Table 6: Rule for Matching Business Entities with Rating Requirement

Textual Format	Jena Rule Format

Business Entity Y
requests an offer from a Business Entity
includes a service which
supports a specific operation Y
	`@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.`
	`@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.`
	`@prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>.`
	`@prefix v1: <http://purl.org/goodrelations/v1#>.`
	`@prefix p1: <http://www.owl-ontologies.com/mason.owl#>.`
	`@prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>.`
	
	`[matchBusinessEntities:`

And Business Entity Y's request has a requested minimum rating for possible supplier Business Entity X offers a service which supports an operation which based on Generic Terms Catalog is mapped with operation Y ***And Business Entity X' rating is greater or equal to demanded minimum rating*** Then Entity Y matches with Entity X	(?y rdf:type v1:BusinessEntity) (?y v1:seeksOffer ?Offery) (?Offery v1:includes ?Servicey) (?Servicey comp:seeksOperation ?Operationy) ***(?Offery comp:hasMinRating ?minRating)*** (?x rdf:type v1:BusinessEntity) (?x MSDL:hasService ?Servicex) (?Servicex comp:hasOperation ?Operationx) (?Operationx comp:mappedToCommonTerm ?Operationy) ***(?x comp:hasRating ?ratingx)*** ***ge(?ratingx, ?minRating)*** ->(?y comp:matchesWith ?x)]

Additional criteria by the requester agent can improve even more the Matchmaker's result. After the initial matching based on the provided services the Matchmaker is able to apply more rules in order to exclude some suppliers from its final output. The rules that will be applied are related to quantitative criteria of the services. For example, a waste management service is capable to handle a limited number of wastes tonnages or a manufacturing service is able to produce a specific number of units/products. The next generic rule is applied for the exclusion of agents (business entities) from the matching ones based on services' capabilities:

Table 7: Jena Rule for Capability Fulfilment

Textual Format	Jena Rule Format
Business Entity X requests an offer which has a quantity requested specification with value quantity X And Business Entity X matches with Business Entity Y Which has s a service with Capability of Value quantity Y If quantity Y is less than quantity X Then drop Business Entity Y from them which matches with Entity X	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>. @prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [capabilityFulfillment: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx v1:hasEligibleQuantity ?QuantitySpecx) (?QuantitySpecx v1:hasValue ?Quantityx) (?x comp:matchesWith ?y) (?y MSDL:hasService ?Servicey) (?Servicey MSDL:hasCapability ?Capabilityy) (?Capabilityy v1:hasEligibleQuantity ?QuantitySpecy) (?QuantitySpecy v1:hasValue ?Quantityy) lessThan(?Quantityy, ?Quantityx) ->drop(4)]

Besides the matchmaking on common terms services catalogue, the semantic matchmaker performs matching also in material level in order to provide accurate matchmaking for possible customers especially for the cases of the waste management providers and raw material suppliers. The following scenario explains better the *find possible customers'* functionality.

- COMPOSITION Marketplace contains Companies A, B, C which are manufacturers and Companies E and F which are waste management providers.
- Company D is a new waste management company at the Marketplace
- Company D collects and manage a wide catalogue of materials
- Company D wants to find possible customers at the Marketplace in order to advertise their services

Problem: It is not so useful for Company D to advertise its services in other waste management companies or to manufacturers that do not work with materials that Company D is able to handle

Solution: The Matchmaker capitalizes on information related to machine processes and materials in order to provide an effective matching for participants who search for new customers in the Marketplace. The semantic rules explores the manufacturing services which are associated with machines and tools, and they are usable on specific materials in order to perform an efficient matchmaking

As depicted in Figure 7, the Matchmaker is able to match the Company D only with the Companies A and B which are possible new customers for the Company D. By applying the rule which is described in Table 8 the Matchmaker returns to the requester an optimal list of possible future customers that is not contains other companies of the same domain (actually, they are competitors) or manufacturers that do not produce wastes able to be handled by the requester.

Table 8: Rule for Finding Possible Customers

Textual Format	Jena Rule Format

Business Entity Y
has a service supports an operation
that is related to a material Y
Business Entity X
has s a service with an operation that
requires a machine which uses a tool
This tool is usable on a material X
which based on Generic terms
catalogue is mapped to material Y
 	
Then Entity Y matches with Entity X
X
	`@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.`
	`@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.`
	`@prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>.`
	`@prefix v1: <http://purl.org/goodrelations/v1#>.`
	`@prefix p1: <http://www.owl-ontologies.com/mason.owl#>.`
	`@prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>.`
	
	`[matchBusinessEntities:`
	
	`(?y rdf:type v1:BusinessEntity)`
	`(?y MSDL:hasService ?Servicey)`
	`(?Servicey comp:hasOperation ?Operationy)`
	`(?Operationy p1:allowedProcessFor ?materialy)`
	`(?x rdf:type v1:BusinessEntity)`
	`(?x MSDL:hasService ?Servicex)`
	`(?Servicex comp:hasOperation ?Operationx)`
	`(?Operationx p1:requiresMachine ?machinex)`
	`(?machinex p1:usesTool ?toolx)`
	`(?toolx p1:toolUsableOn ?materialx)`
	`(?materialy comp:mappedToCommonMaterial ?materialx)`
	`->`
	`(?y comp:matchesWith ?x)`
	`]`

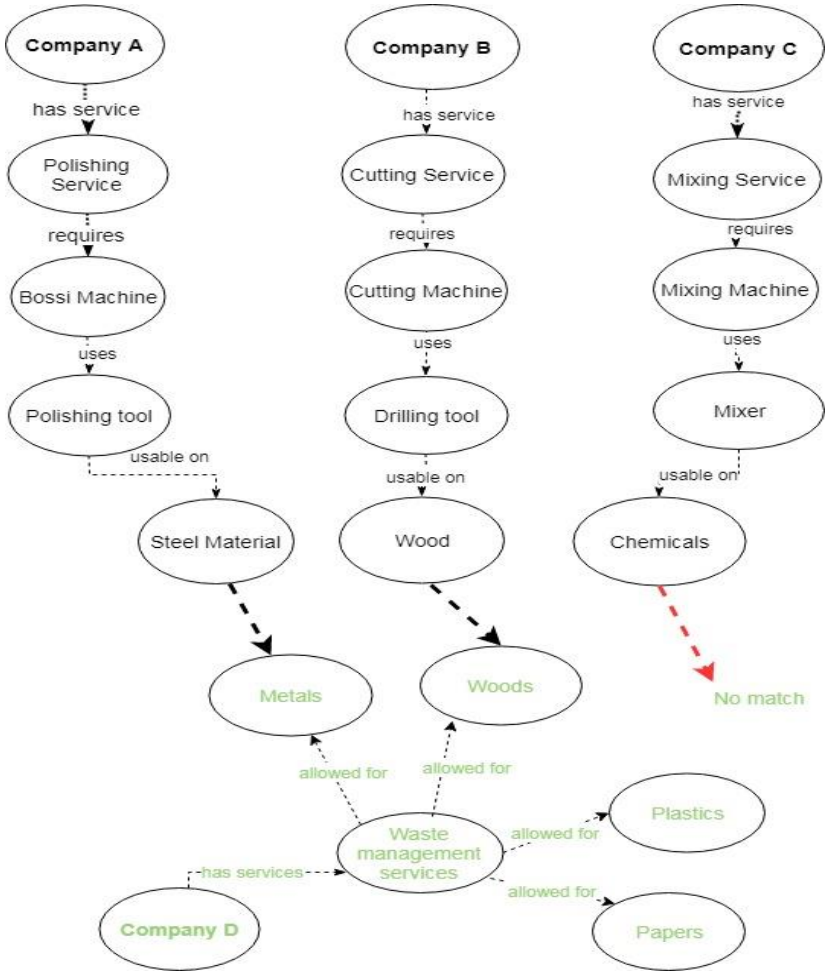

Figure 7: Find Possible Customers Based on Materials Capability

Offer Level Matchmaking

The Offer Level Matchmaking module is related to offers' evaluation. A Marketplace agent can provide to the Matchmaker a set of offers that this agent had received from supplier agents in order to ask for offers' evaluation. Based on this feature, the Matchmaker can act as Broker who aims to match the needs of the requester agents with the best available offer based on different kind of criteria.

In this level of matchmaking the sets of rules were designed for quantitative values' comparisons and evaluation. The algorithm which is followed is a kind of an elimination process in which the instances that do not fulfil a request's requirement are excluded from the matching set. The rules are constructed in a generic way, in order to provide different evaluation results if they are applied to the same offers but in a different sequence based on requesters' ranked preferences. These ranked preferences are taken into account by the Matchmaker's decisions. For example, for a set of identical offers, a requester, who wants quick delivery over the price, will get a different result by the Matchmaker than a requester who has the price as the top priority. After the matchmaking process, the best matching offer and the corresponding supplier agent are returned to the requester agent.

A pseudocode which explains the steps which are followed and executed in the Offer Level Matchmaking module is presented below:

Table 9: Pseudocode of Offer Level Matchmaking Module

1. Read the provided offers

2. Read the requester's ranked N preferences

3. For every offer
4. create the corresponding ontology instance

5. *Set all the available offers as best available*
6. *Create an ordered list of N rule sets based on the ranked preferences*
7. *For rule sets 1 to N*
8. *Apply Rule set #1 to the Ontology Model and exclude the matching offers which did not fulfil this rule from the best available*

   . . .

   Apply Rule set #N to Ontology Model and exclude the matching offers which did not fulfil this rule from the best available
9. *Return the best available offer*

In order to create a generic matchmaking engine which will be easily used in collaborative manufacturing ecosystems the rules were developed to cover the most important factors in the negotiations and transactions in such ecosystems. Based on research and discussions with the project's pilot partners the most important factors in their transactions are the following:

1. The *price* as in almost any transaction the target of the traders is the maximum profit
2. The *quick delivery* of a product or service. In many cases this factor is of great importance. For example in cases in which the production line is running continuously as there are a lot of orders the quick delivery of raw materials is more important than the price.
3. The *trust*. Before a transaction the requester of a service or product wants to be sure that the supplier is trusted, with good reviews by previous users etc.

Based on these factors the following sets of rules are created. Actually, they are pairs of semantic rules. The logic behind these pairs is that the first rule finds the best available value of a factor, and the second rule excludes the offers that contain values of this factor that do not match to the best one.

Table 10: Find Best Available Price

Textual Format	Jena Rule Format
Business entity requests an Offer X which has a price value X And Offer X matches to Offer Y which has price value Y if this value is less than value X Then the requested Offer X has price value equal to price value Y	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>. @prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [findMinPriceOffer: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx v1:hasPriceSpecification ?PriceSpecx) (?PriceSpecx v1:hasCurrencyValue ?Valuex) (?Offerx comp:bestMatchingOffer ?Offery) (?Offery v1:hasPriceSpecification ?PriceSpecy) (?PriceSpecy v1:hasCurrencyValue ?Valuey) lessThan(?Valuey, ?Valuex) -> drop(3) (?PriceSpecx v1:hasCurrencyValue ?Valuey)]

Table 11: Rule to Match Request to Best Price

Textual Format	Jena Rule Format

Business Entity X requests Offer X Offer X has best available price Value X Offer X matches best with Offer Y Offer Y has price Value Y If Value Y is not equal to best price Value X Then remove Offer Y from the best matching Offers	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>. @prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [matchRequestToBestOfferByPrice: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx v1:hasPriceSpecification ?PriceSpecx) (?PriceSpecx v1:hasCurrencyValue ?Valuex) (?Offerx comp:bestMatchingOffer ?Offery) (?Offery v1:hasPriceSpecification ?PriceSpecy) (?PriceSpecy v1:hasCurrencyValue ?Valuey) notEqual(?Valuey, ?Valuex) -> drop(4)]

The same pairs of rules have been implemented for the other two factors: Delivery time and the Rating of the agents in the Marketplace:

Table 12: Rule to Find Best Available Delivery Time

Textual Format	Jena Rule Format
Business entity requests an Offer X which has a best delivery time value X And Offer X matches to Offer Y which has delivery time value Y if this value is less than value X Then the requested Offer X has best delivery time value equal to price value Y	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>. @prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [bestDeliveryLeadTime: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx v1:hasEligibleQuantity ?deliveryTimex) (?deliveryTimex v1:hasMinValueInteger ?deliveryMax) (?Offerx comp:bestMatchingOffer ?Offery) (?Offery v1:hasEligibleQuantity ?deliveryTimey) (?deliveryTimey v1:hasMinValueInteger ?deliveryMiny) lessThan(?deliveryMiny, ?deliveryMax) -> drop(3) (?deliveryTimex v1:hasMinValueInteger ?deliveryMiny)]

Table 13: Rule to Match Request to Best Delivery Time

Textual Format	Jena Rule Format
	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>.

Business Entity X requests Offer X Offer X has best available delivery time Value X Offer X matches best with Offer Y Offer Y has delivery time Value Y If Value Y is not equal to best Value X Then remove Offer Y from the best matching Offers	@prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [matchToBestDeliveryLeadTime: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx v1:hasEligibleQuantity ?deliveryTimex) (?deliveryTimex v1:hasMinValueInteger ?deliveryx) (?Offerx comp:bestMatchingOffer ?Offery) (?Offery v1:hasEligibleQuantity ?deliveryTimey) (?deliveryTimey v1:hasMinValueInteger ?deliveryMiny) notEqual(?deliveryx, ?deliveryMiny) ->drop(4)]

Table 14: Rule to Find Best Available Rating

Textual Format	Jena Rule Format
Business entity requests an Offer X which has a best available rating value X And Offer X matches to Offer Y which provided by Business Entity Y with rating value Y if this value is greater than value X Then the requested Offer X has best available rating value equal to price value Y	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>. @prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [matchRequestToBestRatings: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx comp:hasMinRating ?minRating) (?Offerx comp:bestMatchingOffer ?Offery) (?y rdf:type v1:BusinessEntity) (?y v1:offers ?Offery) (?y comp:hasRating ?ratingy) greaterThan(?ratingy, ?minRating) ->drop(2) (?Offerx comp:hasMinRating ?ratingy)]

Table 15: Rule to Match Request to Best Rating

Textual Format	Jena Rule Format
Business Entity X requests Offer X	@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>. @prefix comp: <http://www.composition-project/ontologies/COMPOSITIONv01#>. @prefix v1: <http://purl.org/goodrelations/v1#>. @prefix p1: <http://www.owl-ontologies.com/mason.owl#>. @prefix MSDL: <http://www.composition-project.eu/ontologies/MSDL#>. [matchRequestToBestOfferByRating: (?x rdf:type v1:BusinessEntity) (?x v1:seeksOffer ?Offerx) (?Offerx comp:hasMinRating ?minRating)

Offer X has best available rating Value X Offer X matches best with Offer Y Offer Y is offered by Business Entity Y which has rating Value Y If Value Y is not equal to best Value X Then remove Offer Y from the best matching Offers	(?Offerx comp:bestMatchingOffer ?Offery) (?y rdf:type v1:BusinessEntity) (?y v1:offers ?Offery) (?y comp:hasRating ?ratingy) notEqual(?ratingy, ?minRating) ->drop(3)]

As mentioned before these rules are constructed in order to provide different evaluation results if they are applied to the same offers but in a different sequence based on requesters' ranked preferences. In the case that the requester prefers price over delivery time and the Marketplace rating is its last preference the pairs of rules will be applied in the sequence that they are presented above. However, in the case that the requester prefers delivery time as first choice, the rating as the second and the price as the last one then the Matchmaker will execute the rules pair from tables 12, 13 then the rules from tables 14, 15 and last the rules from tables 10, 11. New rules in the same pairs' format can be added to enrich the matchmaking process with more factors.

6.5 Quality Control

This deliverable is part I and the developed software are the first versions of the COMPOSITION Matchmaker. The work in this first part was focused on research and analysis of related work, technologies and tools, the architecture's design and the implementation of a Matchmaker version that is able to support the online bidding processes over the Marketplace. A quality control plan has been followed during the development processes.

During the implementation phase of COMPOSITION Matchmaker the quality control was focused on general software quality criteria, the overall COMPOSITION system architecture's compatibility and the deliverable D1.1 Project Quality Control Plan I of COMPOSITION project. More precisely the quality plan consists of the following factors:

- Identification of the requirements related to the Matchmaker
- Analysis of existing technologies and adoption of the best suitable with the COMPOSTITION system's architecture. Use of REST web services and JSON format for messages exchange as both technologies have defined as supported by COMPOSITION architecture at D2.3-The COMPOSITION architecture specification I. These will ensure the compatibility with other project's components.
- Use of software tools which were proposed at D1.1 Project Quality Control Plan I and support quality of software:
    - Use of Eclipse IDE (ECLIPSE IDE, 2018) as the development environment
    - Use of Git for control versioning. The EGit (ECLIPSE EGit, 2018) plugin from Eclipse IDE was used.
    - Use of Maven (Apache Maven, 2018) as build tool for dependency management and build of source code
    - Use of Docker (Docker, 2018) for deployment
- Test procedures were applied. For software quality assurance both static and dynamic analysis techniques applied:

  In *static analysis* the PMD (PMD, 2018) tool was used. It is an open source tool which offers source code analysis and offered as an Eclipse IDE plugin. It is able to detect possible bugs, empty statements, unused variables and methods, duplicate code, classes with high cyclomatic complexity etc. by offering built-in sets of rules. The tool categorizes the possible problems as violations distributed in 5 categories based on priority: block, critical, urgent, important and warning

  Almost 300 rules from 20 different rules sets were used. The rules sets were the following: Basic, Basic POM, Braces, Code size, Complexity, Controversial, Design, Empty code, Import statements, J2EE, Junit, Naming, Optimization, Security code guidelines, Strict Exceptions, String & StringBuffer, Style, Unnecessary and Unused code.

The analysis results were evaluated during the development face and the most important possible bugs were handled. At the current version of code there are no block, critical, important and warning violations. Currently there are only few urgent violations which are related to excessively long variable names, variables with short names, multi occurrences of some string literals etc. These violations are considered as false positives.

In *dynamic* analysis, tests in runtime have been executed. Generally in dynamic analysis Unit tests, Integration tests and System tests should be executed.

We built automated tests in source code package which was created by Maven. The TestCase class from JUnit was extended and member functions were added. Every function represents a test of a supported web service. The tests are able to be executed without deploying the project at an external and using an external HTTP client. We used Eclipse Jetty server which provides a Web server and javax servlet container. So, the test cases deployed and executed using Jetty. This provided us fast execution and testing of the source code without the need to deploy the project to an external server in order to test every change in the code. The tests were called both separately or in combination.

Moreover, the Matchmaker was tested in integration with the Marketplace agents. Both Matchmaker and agents were deployed as Docker images. The Agents image was able to call successfully the Matchmaker's APIs services from the deployed image as well. Furthermore, the correctness of the Matchmaker responses was checked too. More details about agents and Matchmaker interaction, Matchmaker APIs and the deployment of the component are presented to the following chapter.

- The performance of the Matchmaker was also tested. Based on the project requirements the Matchmaker should respond within a reasonable time frame (e.g. 5 seconds). The current version of the Matchmaker returns its output within 1-2 seconds. A better picture about the Matchmaker performance will be available by the end of the Task 6.5 as by then the Marketplace will contain more individuals and the matchmaking process will be more challenging in the terms of performance.

The complete quality plan will be presented at the next and final version of this document and it will represent the complete quality process which is followed until the final delivered version of the Matchmaker.

7 Matchmaker APIs and Deployment

In this chapter the services offered by the Matchmaker API are presented. Furthermore, the deployment of the Matchmaker component is presented as well.

7.1 Matchmaker API Web Services

The Matchmaker is connected with the Marketplace agents through RESTful web services and HTTP protocol. An API is offered to the agents. The implemented Matchmaker API contains two web services. As depicted in Figure 8 below, both of the offered web services are POST requests

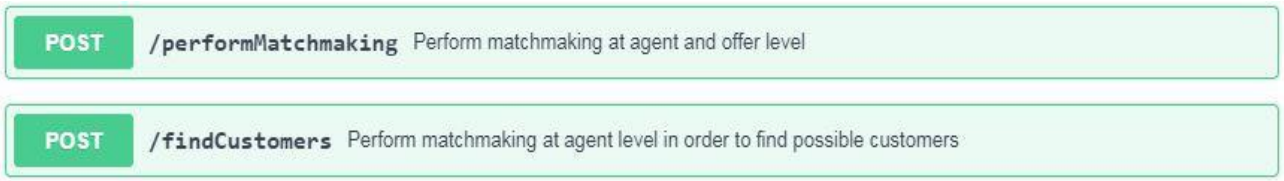

Figure 8: Matchmaker API Services

performMatchmaking

This web service was designed in order to support the online bidding processes over the COMPOSITION Marketplace. Based on the request's body the matchmaker decides if it is going to apply Offer Level or Agent Level matchmaking as both are required in a bidding process. The collaboration scheme of the agents and the Matchmaker presented to the following figure:

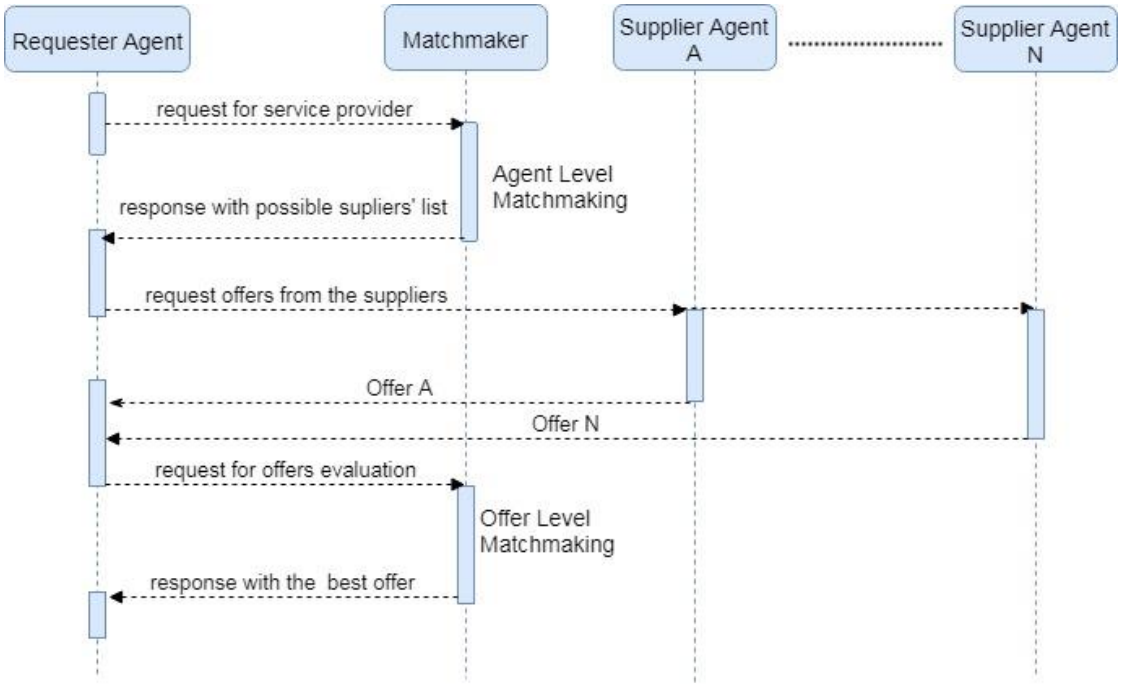

Figure 9: Matchmaker and Agents Communication

As mentioned before the request's body defines the type of the matchmaking which will be triggered. The body is defined in JSON in a format compatible with the agents' CXL. In order to trigger the Agent Level matchmaking the following body is posted to the Matchmaker:

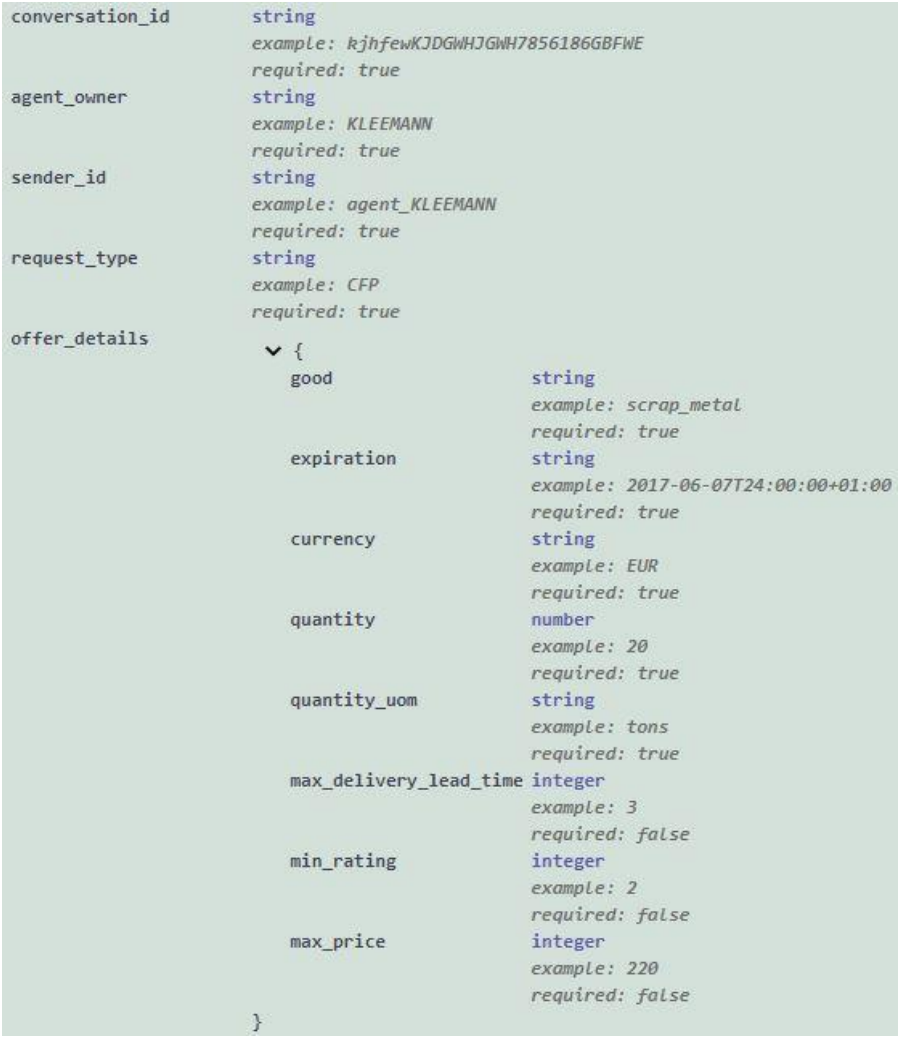

Figure 10: Request Body for Agent Level Matchmaking

In the abovementioned presented request's body:

- *conversation_id* is the unique id of the conversation allows to track request / reply sequences
- *agent_owner* describes the business entity's agent generating the message
- *sender_id* is the id of the requester agent
- *request_type* describes the type of the request and defines the level of matchmaking(Agent or Offer)
- *offer_details* object describes the details of the request such as the good/service type that is requested, the expiration date of the request, the requested quantity and its corresponding unit of measurement. The last three values are optional and describe additional requirements that the requester can set in order to receive a more accurate matchmaking result based on services capabilities were described at the Collaborative Manufacturing Services Ontology.

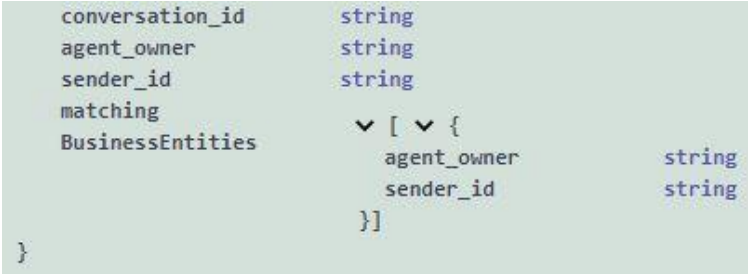

Figure 11: Response of Agent Level Matchmaking

In order to call the Offer Level of matchmaking in the request body the request_type property is set as “OFFER”. Moreover, an array containing the offers which were provided by the supplier agents is added to the body object. The ranked preferences of the Offer Level matchmaking are added as well. A figure that describes this type of request’s body is provided at the ANNEX of the current report.

findCustomers

This web service is designed in order to enable agents to find possible customers for their services in the COMPOSITION Marketplace. This functionality is related to Atlantis use case scenarios in which the Marketplace should offer solutions to its participant. This service offers the opportunity to the Marketplace participants to advertise its services and products to possible customers. The functionality of this service was presented in more details at chapter 6 of this report. The request’s body schema is presented to the following figure:

```
conversation_id      string
                     example: kjhfewKJDGWHJGWH7856186GBFWE
                     required: true
agent_owner          string
                     example: ELDIA
                     required: true
sender_id            string
                     example: agent_ELDIA
                     required: true
request_type         string
                     example: CFP
                     required: true
```

Figure 12: Request Body for findCustomers Service

The response of the findCustomers services has the same schema as the performMatchmaking services presented at Figure 11.

7.2 Matchmaker Deployment

The Matchmaker component was decided to be deployed as a Docker image as the rest of the project’s component based on the Deployment View of D2.3 The COMPOSITION architecture specification I.

Docker is an open-source project aiming at automating the deployment of applications as portable, self-sufficient containers that can run virtually anywhere, on any kind of server. It can be considered as a lightweight alternative to full machine virtualization provided by hypervisors. While in the traditional hypervisor approaches each virtual machine (VM) needs its own operating system, in Docker applications operate inside a container that resides on a single host operating system that can serve many different containers at the same time.

The Matchmaker’s Docker image contains the complete component as it is described at Figure 1 at chapter 4. So in this image the Rule-based Matchmaker, the Query Engine, the Ontology Store and their corresponding APIs are containing.

In order to create the Matchmaker’s Docker image and the corresponding container the official Docker image for Apache Tomcat (Apache Tomcat, 2018) was used. Tomcat was selected as the web server environment as it is web server environment in which Matchmaker’s Java code can run. So, for the creation of the aforementioned Docker image the Web Application Resource file from the Matchmaker was added to the Tomcat’s image. The corresponding Docker container of the Matchmaker image was deployed at the COMPOSITION inter-factory Portainer (Docker, 2018) which offers management of Docker environments. A view of the COMPOSITION inter-factory Portainer which is related to Marketplace components presented at Figure 13.

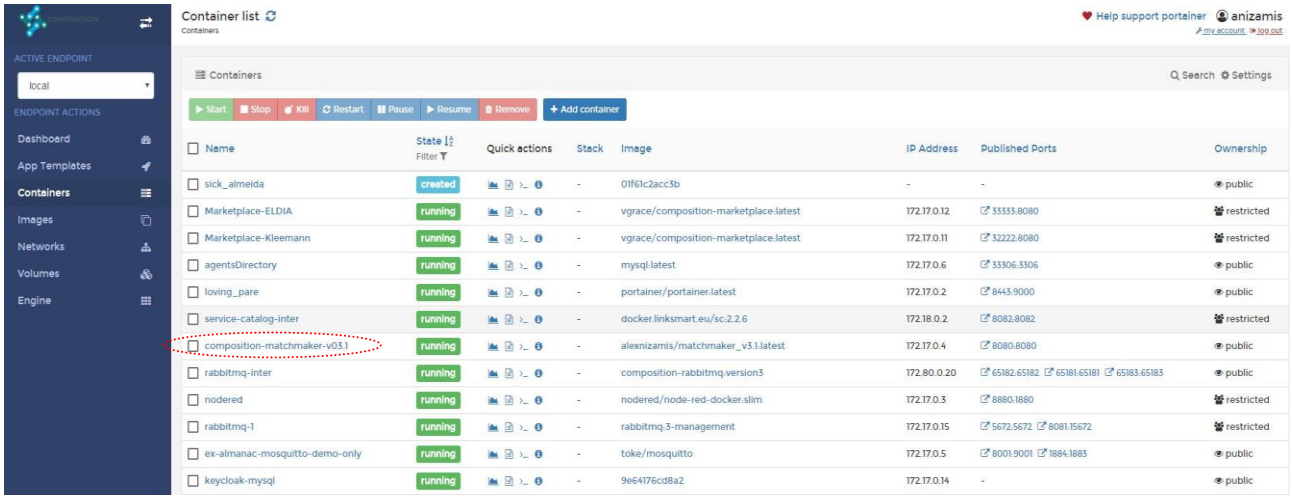

Figure 13: COMPOSITION Inter-factory Portainer

8 Next steps

The future work at Task 6.5 Brokering and Matchmaking for Efficient Management of Manufacturing Processes will be mainly focused at procedures related to:

- The continuously update of the Matchmaker in order to be compatible with the Collaborative Manufacturing Services Ontology's changes. As the functionality of the Matchmaker is exclusively related to the Ontology any modification to the Ontology may lead to modifications in Matchmaker's rules. Moreover, extension in Ontology may offer more intelligence to the Matchmaker by constructing new rules based on new offered information and data descriptions which will be related to manufacturing processes and resources.
- The extension of the Matchmaker's capabilities in order to support ATL's uses case scenarios which is related to software solutions and consulting. Of course, this extension requires the corresponding extension at the Collaborative Manufacturing Services Ontology as well.
- The further extension of the semantic rules in order to contain more factors. This will increase the precision of the matchmaking output and will offer a more accurate result to the Marketplace agents.
- The continuously update and extension of the Matchmaker API in order to provide access to the new matchmaking services to the Marketplace agents.
- A Keycloak (Keycloak, 2018) adapter will be installed in Tomcat server in which the Matchmaker component is deployed in order to be compatible with project's security requirements and the implemented Security Framework from WP4

9 Summary and conclusions

In conclusion, this deliverable describes the effort spent from M5 to M20 and represents the current status of Task 6.5 - Brokering and Matchmaking for Efficient Management of Manufacturing Processes of WP6. Moreover, this report documents the implemented COMPOSITION Matchmaker. The complete work of Task 6.5 will be presented in D6.10 COMPOSITION Brokering and Matchmaking components I II in M34.

The COMPOSITION Matchmaker has been implemented and presented after an analysis of related works, and available tools and technologies. The implemented Collaborative Manufacturing Services Ontology from Task 6.4 is also presented as the Matchmaker's functionality is exclusively depended on this ontology. Moreover the implemented version of the Matchmaker was presented in this report with emphasis at semantic rules as it is a rule-based matchmaker which infer new knowledge by applying rules.

After consideration of project's requirements and architecture, and after an analysis of available technologies and tools, a Matchmaker API is developed in Java and it is offered through RESTful web services. It provides to the Marketplace agents access to the matchmaking functionalities.

A working version of the Matchmaker component which contains the Rule-based Matchmaker and its corresponding API, the Ontology API and the Ontology Store has been deployed as a Docker container in the COMPOSIITON inter-factory container. This deployment enables the usage of these components by the Marketplace agents.

The outcome of this deliverable mainly affects the WP6 and its components, the agents. By using the Matchmaker services the agents are able to execute automated bidding processes in the Collaborative Ecosystem or to find possible future customers within this ecosystem.

Finally, as it is perceived, the first steps of Task 6.5 are presented in this deliverable. However, the work has been done should be further extended, as it described at Chapter 8 - Next Steps. More functionalities should be added at the Matchmaker in order to be able to support all the COMPOSITION use cases which are related to this component.

10 List of Figures and Tables

10.1 Figures

10.2 Tables

11 References

(Sycara, 1999) Sycara, K., Klusch, M., Wido, S., and Lu, J. (1999). Dynamic Service Matchmaking Among Agents in Open Information Environments, volume 28, 47-53. ACM, New York, NY, USA.

(Nodine, 2000) Nodine M., Fowler, J., Ksiezyk, T., Perry, B., Taylor, M., and Unruh, A. (2000). Active information gathering in InfoSleuth. International Journal of Cooperative Information Systems, 9(1/2)

(Ameri, 2012) Ameri, F. and Patil, L. (2012). Digital manufacturing market: a semantic web-based framework for agile supply chain deployment. Journal of Intelligent Manufacturing, 23(5), 1817-1832.

(Ameri, 2006) Manufacturing Service Description Language https://www.researchgate.net/publication/267486591_An_Upper_Ontology_for_Manufacturing_Service_Description

(Lemaignan, 2006) Manufacturing's Semantics Ontology or MASON is a manufacturing ontology, aimed to provide a common semantic net in manufacturing domain. http://ieeexplore.ieee.org/document/1633441/

(M. C. Suárez-Figueroa, 2010) NeOn Methodology for Building Ontology Networks: Specification, Scheduling and Reuse

(FIWARE, 2018) Retrieved from FIWARE: http://www._ware4industry.com/

(GoodRelations Language, 2018) Retrieved from GoodRelations Language: http://www.heppnetz.de/projects/goodrelations

(Apache Jena, 2018) Retrieved from JENA: https://jena.apache.org/documentation/inference/

(Apache Maven, 2018) Apache Maven: https://maven.apache.org/

(Docker, 2018) Docker: https://www.docker.com/

(Apache Maven, 2018) Apache Maven: https://maven.apache.org/

(PMD, 2018) PMD: https://pmd.github.io/

(Apache Tomcat, 2018) Apache Tomcat: http://tomcat.apache.org/

(Keycloak, 2018) Retrieved from Keycloak: https://www.keycloak.org/

(ECLIPSE EGit, 2018) Retrieved from ECLIPSE: http://www.eclipse.org/egit/

(ECLIPSE IDE, 2018) Retrieved from ECLIPSE: https://www.eclipse.org/ide/

(FITMAN-SeMa, 2018) Retrieved from http://www.ware4industry.com/?portfolio=metadata-and-ontologies-semantic-matching-sema/

ANNEX

Matchmaker Offer Level Request's Body:

```
conversation_id        string
                       example: kjhfewKJDGWHJGWH7856186GBFWE
                       required: true
agent_owner            string
                       example: KLEEMANN
                       required: true
sender_id              string
                       example: agent_KLEEMANN
                       required: true
request_type           string
                       example: CFP
                       required: true
offer_details          { 
                         good                  string
                                               example: scrap_metal
                                               required: true
                         expiration            string
                                               example: 2017-06-07T24:00:00+01:00
                                               required: true
                         currency              string
                                               example: EUR
                                               required: true
                         quantity              number
                                               example: 20
                                               required: true
                         quantity_uom          string
                                               example: tons
                                               required: true
                         max_delivery_lead_time integer
                                               example: 3
                                               required: false
                         min_rating            integer
                                               example: 2
                                               required: false
                         max_price             integer
                                               example: 220
                                               required: false
                       }
offers                 [ { 
                         offer_details
                                               { 
                                                 agent_owner           string
                                                                       example: ELDIA
                                                                       required: true
                                                 sender_id             string
                                                                       example: agent_ELDIA
                                                                       required: true
                                                 good                  string
                                                                       example: scrap_metal
                                                                       required: true
                                                 currency              string
                                                                       example: EUR
                                                                       required: true
                                                 quantity              number
                                                                       example: 20
                                                                       required: true
                                                 quantity_uom          string
                                                                       example: tons
                                                                       required: true
                                                 delivery_lead_time    integer
                                                                       example: 3
                                                                       required: true
                                                 price                 integer
                                                                       example: 220
                                                                       required: true
                                               }
                        }]
preferences            { 
                         priority_1            string
                                               example: rating
                                               required: true
                         priority_2            string
                                               example: price
                                               required: true
                         priority_3            string
                                               example: delivery_lead_time
                                               required: true
                       }
```