D4. "Prototype demonstration" Full design specification

POC4COMMERCE

01/10/2021

Grant Agreement No.: 957338
Call: H2020-ICT-2020-1

Topic: ICT-54-2020
Type of action: RIA

POC4COMMERCE

D4. "Prototype demonstration" Full design specification

DUE DATE	01/10/2021
SUBMISSION DATE	01/10/2021
TEAM	POC4COMMERCE
VERSION	1.0
AUTHORS	Giampaolo Bella, Domenico Cantone, Cristiano Longo, Marianna Nicolosi Asmundo, Daniele Francesco Santamaria

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

List of Figures

List of Tables

Abbreviations

Executive Summary

Semantic Blockchain is a vision of blockchain where transactions carry semantic annotation describing the meaning and the motivation of the transaction, explaining what, when, and why a transaction is stored in the public ledger. Even though the concept of semantic blockchains is not novel idea, for instance the BLONDiE [5] ontology provides a general semantic description of the meta-information of transactions such as block number, wallet issuer, and transaction fees, an innovative semantic descriptions of smart contracts and of the related transactions, in particular of smart contracts related with tokens trading and associated with commercial means, lies at the core of POC4COMMERCE project.Indeed, POC4COMMERCE aims to deliver a set of well-assessed ontologies for commercial participants, blockchains, and commercial offers, conjoining the high level of interoperability mandatory for the new generations of WEB 3.0 trustworthy commercial applications. To practically consume the ontological stack, a prototype of a semantic search engine is also delivered as to concretely probe

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

and reason on the semantic blockchain, and to assist service and product providers to generate its own data and offerings to be published, and user to share their commercial experience.

The POC4COMMERCE Semantic Search Engine is constituted by two principal modules: the OCGEN module and the OCCSE module. The OCGEN module provides APIs to deal with semantic web agents, in particular, to generate ONTOCHAIN agents, contributors, and related, in the OC-Found ontology fashion, whereas the OCCSE module is constituted by several modules and APIs enabling users for accessing the search engine functionalities, including reasoning techniques, storage servinces interface, and query mechanisms.

The ontological stack of POC4COMMERCE as been deeply described in Deliverable 3, whereas this document is focused on the description of the semantic search engine, drawing some future expectations and ideas. Finally, we also depict the case study of the iExec marketplace in the POC4COMMERCE ontological stack framework.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Contents

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

1 Technical Results

1.1 Project Repositories

The POC4COMMERCE ontological stack and the OCCSE search engine are available at https://github.com/ONTOCHAIN/POC4COMMERCE. Specifically, the repository is organized as follows: the folder *ontologies* contains the ontologies delivered within the ontological stack; the folder *python* contains the source code and the test files of the OCCSE search engine. Inside the folder *python*, there are three folders, namely, *OCCSE* containing the library for the search engine module, the folder *OCGEN* contains the library for the ontology generator SDK. Finally, folder *test* contains file and data examples.

Repository Name	Link	Kind	Eventual Restrictions
POC4COMMERCE repository	https://github.com/ONTOCHAIN/ POC4COMMERCE	ONTOCHAIN	

1.2 Program Modules Overview

POC4COMMERCE delivers a stack of three ontologies, namely, OC-Found, OC-Commerce, and OC-Ethereum, and an API library consisting of two main modules:

- **OC-Generator**, in short OCGEN, providing basic APIs for generating agent ontological representations in the mentalistic notion of agent behaviors fashion, adopted by the foundational ontology OC-Found.
- **OC-Commerce Search Engine**, in short OCCSE, providing APIs that realize the core of a semantic search engine and reasoning system for querying the ONTOCHAIN knowledge bases.

The modules OCGEN and OCCSE have been implemented in Python, version 3.7. Specifically, OCGEN adopts the RDFLib[1] version 5 API library[2] to generate RDF fragments, whereas OCCSE adopts the API library OWLReady 2[3] version 0.34 to realize the query engine. Whereas RDFLib provides low-level APIs for generating RDF triples in an extremely efficient way, OWLReady 2 provides an higher level interface for integrating OWL reasoners and performing SPARQL queries compliant with the latter version 1.1.

[1]https://pypi.org/project/rdflib/5.0.0/
[2]RDFLib version 6 has currently a bug preventing the OCGEN prototype from loading remote ontologies.
[3]https://pypi.org/project/Owlready2/

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Name	Description	Language	Kind
OCGEN	This module assists the user to generate RDFs about POC4COMMERCE ontological stack	Python	API
OCCSE	This module assists the user to query and reason on POC4COMMERCE knowledge base	Python	API

The architecture of OCGEN is monolithic, since it must provide basic functionalities for generating RDF graph, whereas the architecture diagram of the OCCSE module is depicted in Figure 1.

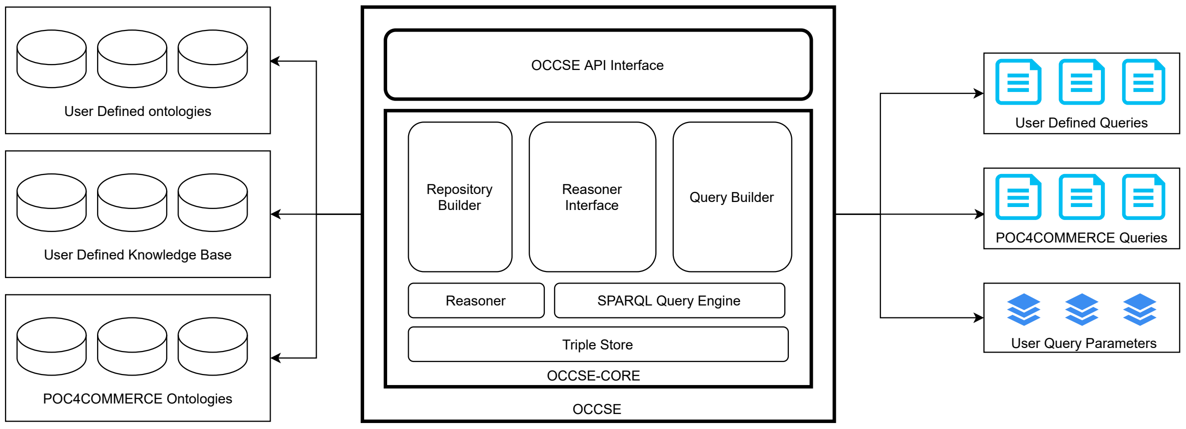

Figure 1: OCCSE architecture diagram

The OCCSE search engine provides API interface allowing user to access the search engine functionalities. The core of OCCSE comprises six modules:

- *Repository Builder*. This module permits to fetch and access ontologies available both on local and remote storage systems.
- *Reasoner Interface*. This module allows the connection between a reasoner such as HermiT and Pellet and the RDF triple-store.
- *Query Builder*. This module allows users to create and perform both custom SPARQL queries and queries derived from the competency questions defined for the POC4COMMERCE ontological stack. The module also admits parametric queries, namely, queries admitting variables selected by users on run-time.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

- *SPARQL query engine*. This module is derived from the OWLReady 2 API library and permits to execute SPARQL queries defined by users over the data stored in the local RDF triple store.
- *Reasoner*. This module allows one to perform reasoning and to infer new knowledge. Users may select a reasoner through the *Reasoner Interface* module.
- *Triple Store*. This module permits to store data selected by users through the *Repository Builder* module.

1.3 APIs for SDKs

In this section we introduces the APIs provided by the OCGEN and OCCSE modules. The OCGEN APIs comprise tools for generating RDF graphs for the POC4COMMERCE knowledge base, in particular, for the definition of agents in the OC-Found fashion, whereas the OCCSE APIs provides the tools for reasoning on and querying of RDF graphs.

1.3.1 The APIs of OCGEN

The OCGEN main object can be instantiated first by creating three RDFLib ontology objects, one for the ontology hosting the agent behaviors, one for the ontology hosting the agent templates, one for the ontology hosting data, then by calling the constructor of the class *OCGEN*, namely *OCGEN(ontology, namespace, ontologyURL, ontologyTemplate, namespaceTemplate, templateURL)* where

- *ontology*, of type RDF graph is the ontology hosting the agent behaviors followed.
- *namespace*, of type string, is the namespace of *ontology*.
- *ontologyURL*, of type string, is the URL of *ontology*.
- *ontologyTemplate*, of type RDF graph, is the ontology hosting the agent templates.
- *namespaceTemplate*, of type string, is the namespace of *ontologyTemplate*.
- *templateURL*, of type string, is the URL of *ontologyTemplate*.

The module will store RDF graphs in the correct ontology depending on the type of operation required. It is also possible to create or load one single ontology for storing both behaviors, templates, and data.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

The OCGEN module provides several methods to deal with semantic web agents in the OASIS ontology fashion:

- *createAgentTemplate(agentTemplateName)* is a void method creating an agent template given *agentTemplateName* as input parameter, namely a string representing the agent template name.
- *createAgentBehaviorTemplate(MyTemplateBehavior, MyTemplateGoal, MyTemplateTask, [MyTemplateTaskOperator, action], [MyTemplateOperatorArgument, actionArgument], [[MyTemplateTaskObject, taskObjectProperty, objectTemplate]], [[MyTemplateInput1, taskInputProperty, input1]], [[MyTemplateOutput1, taskOutputProperty, output1]])*, is a void method creating a behavior template to be associated with an agent template, where:
  - *MyTemplateBehavior* is a string representing the entity name of the behavior template.
  - *MyTemplateGoal* is a string representing the entity name of the goal template.
  - *MyTemplateTask* is a string representing the entity name of the task template.
  - *MyTemplateTaskOperator* is a string representing the entity name of the task operator.
  - *action* is a string representing the operator action.
  - *MyTemplateOperatorArgument* is a string representing the entity name of the operator argument.
  - *actionArgument* is a string representing the operator argument as defined in OASIS-ABox.
  - *[[MyTemplateTaskObject, taskObjectProperty, objectTemplate]]* is a list of elements where:
    - *MyTemplateTaskObject* is a string representing the entity name of the task object.
    - *taskObjectProperty* is either the string *refersAsNewTo* or *refersExactlyTo*.
    - *objectTemplate* is a string representing the element associated to the task object.
  - *[[MyTemplateInput1, taskInputProperty, input1]]* is a list of elements where:
    - *MyTemplateInput1* is a string representing the entity name of the input.
    - *taskInputProperty* is either the string *refersAsNewTo* or *refersExactlyTo*.
    - *input* is a string representing the element associated to the input element.
  - *[[MyTemplateOutput1, taskOutputProperty, output1]]* is a list of elements where:
    - *MyTemplateOutput1* is a string representing the entity name of the output.
    - *taskOutputProperty* is either the string *refersAsNewTo* or *refersExactlyTo*.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

        - *output* is a string representing the element associated with the output element.

- *connectAgentTemplateToBehavior(MyAgentBehaviorTemplate, MyTemplateBehavior)*, is a void method associating a behavior template to an agent template, where
    - *MyAgentBehaviorTemplate* is a string representing the behavior template created as described above.
    - *MyTemplateBehavior* is a string representing the behavior created as described above.
- *createAgent(MyAgent)*, is a void method creating a real agent where *MyAgent* is a string representing the agent name.
- *createAgentBehavior(MyAgentBehavior, MyAgentGoal, MyAgentTask, [MyAgentTaskOperator, action], [MyAgentOperatorArgument, actionArgument], [[MyAgentTaskObject, taskObjectProperty, agentobject1]], [[MyAgentInput1, taskInputProperty, agentinput1]], [[MyAgentOutput1, taskInputProperty, agentoutput1]], [MyTemplateTask, [[MyAgentTaskObject, MyTemplateTaskObject]], [[MyAgentInput1, MyTemplateInput1]], [[MyAgentOutput1, MyTemplateOutput1]]])*, is a void method creating a concrete behavior to be associated with a concrete agent, where
    - *MyAgentBehavior* is a string representing the entity name of the behavior.
    - *MyAgentGoal* is a string representing the entity name of the goal.
    - *MyAgentTask* is a string representing the entity name of the task.
    - *MyAgentTaskOperator* is a string representing the entity name of the task operator.
    - *action* is a string representing the operator action as defined in OASIS-ABox.
    - *MyAgentOperatorArgument* is a string representing the entity name of the operator argument.
    - *actionArgument* are, respectively, and the operator argument as defined in OASIS-ABox.
    - *[[MyAgentTaskObject, taskObjectProperty, agentobject1]]* is a list of elements where
        - *MyAgentTaskObject* is a string representing the entity name of the task object.
        - *taskObjectProperty* is the either the string *refersAsNewTo* or *refersExactlyTo*.
        - *agentobject1* is a string representing the element associated to the task object.
    - *[[MyAgentOutput1, taskOutputProperty, agentoutput1]]* is a list of elements where

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

        * *MyAgentOutput1* is the entity name of the output.
        * *taskOutputProperty* is either the string *refersAsNewTo* or *refersExactlyTo*.
        * *agentoutput1* is a string representing the element associated to the output element.
    - *MyTemplateTask* is a string representing the task object of the behavior template.
    - *[MyAgentTaskObject, MyTemplateTaskObject]* is a list of elements where *MyAgentTaskObject*, *MyTemplateTaskObject* are two strings representing the entity name of the agent task object and the entity of the task object template, respectively.
    - *[MyAgentInput1, MyTemplateInput1]* is a list of elements where *MyAgentInput1*, *MyTemplateInput1* represent the entity name of the agent input and the agent input template, respectively.
    - *[MyAgentOutput1, MyTemplateOutput1]* is a list of elements where *MyAgentOutput1*, *MyTemplateOutput1* represent the entity name of the agent output and the agent output template, respectively.

- *connectAgentToBehavior(MyAgent, MyAgentBehavior)*, is a void method associating a concrete behavior to a concrete agent, where *MyAgent* and *MyAgentBehavior* are, respectively, the agent and the agent behavior.

- *createAgentAction(MyAgent, planExecution, executionGoal, executionTask, [executionOperator, action], [executionArgument, argument], [[executionObject, taskObjectProperty, executionobject1]], [[executionInput1, inputProp, executioninput1]], [[executionOutput1, outputProp, executionOutput1]], [MyAgentTask, [[executionObject, MyAgentTaskObject]], [[executionInput1, MyAgentInput1]], [[executionOutput1, MyAgentOutput1]]])*, is a void method creating an agent action that is associated with an agent behavior where

    - *MyAgent* is the entity name of the agent responsible for the execution of the action.
    - *planExecution* is the entity name of the plan execution.
    - *executionGoal* is the entity name of the goal execution.
    - *executionTask* is the entity name of the task execution.
    - *[executionOperator, action]* is a list of elements where
        * *executionOperator* is the name of the task operator.
        * *action* is name of the action as defined in OASIS-ABox.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

- *[executionArgument, argument]* is a list of elements where
  - *executionArgument* is the name of the task argument.
  - *argument* is the name of the argument as defined in OASIS-ABox.
- *[executionObject, taskObjectProperty, executionobject1]* is a list of elements where
  - *executionObject* is the entity name of the task execution object.
  - *taskObjectProperty* is either *refersAsNewTo* or *refersExactlyTo*.
  - *executionobject1* is the element associated with the task execution object.
- *[executionInput1, inputProp, executioninput1]* is a list of elements where
  - *executionInput1* is the entity name of task input.
  - *inputProp* is either *refersAsNewTo* or *refersExactlyTo*.
  - *executioninput1* is the element associated with the task input.
- *[executionOutput1, outputProp, executionOutput1]* where
  - *executionOutput1* is the entity name of task output.
  - *outputProp* is either *refersAsNewTo* or *refersExactlyTo*.
  - *executionOutput1* is the element associated with the task output.
- *MyAgentTask* is the task of the agent behavior.
- *[executionObject, MyAgentTaskObject]* is a list of elements where *executionObject*, *MyAgentTaskObject* represent the entity name of the task execution and the entity name of the task object of the agent behavior, respectively.
- *[executionInput1, MyAgentInput1]* is a list of elements where *executionInput1*, *MyAgentInput1* represent the entity name of the action input and the agent behavior input , respectively.
- *[executionOutput1, MyAgentOutput1]* is a list of elements where *executionOutput1*, *MyAgentOutput1* represent the entity name of the action output and the agent behavior output, respectively.

Other methods derived from the RDFLib APIs allow users to define their own RDF graphs.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

1.3.2 The APIs of OCCSE

Before instantiating the OCCSE, a repository manager and a reasoner interface are required. To create a repository manager it is sufficient to create an object of type *RepositoryManager*, passing a list of IRIs representing the repositories that will be loaded into the OCCS triple store. Specifically,

- *RepositoryManager([repository1, repository2, ...]* returns the object *RepositoryManager*, where [repository1, repository2, ...] is a list of IRIs representing the repository to load.

Likewise, the reasoner interface is instantiated by creating an object of type *ReasonerInterface* passing the name of the selected reasoner. Currently, the HermiT and PelleT reasoners are natively supported, but other reasoners can be added in the future. Specifically,

- ReasonerInterface(reasonerName) returns the *ReasonerInterface*, where *reasonerName* is either *pellet* or *hermit* values.

The OCCSE can be finally instantiated by calling the constructor of the class *OCCSE*, passing both the previously created repository manager and the reasoner interface. Specifically,

- *OCCSE(repositoryManager, reasonerInterface)* instantiates the OCCSE module, where *repositoryManager* and *reasonerInterface* are the repository manager and the reasoner interface, respectively, as created before.

In the next steps, the OCCSE will load the selected data on the triple store, synchronize the reasoner, and prepare to perform queries.

To load data on the triple store it is sufficient to call the void method *loadRepository*, whereas the synchronization of the reasoner can be performed by calling the method *syncReasoner*. Moreover, additional prefixes can be bounded to the OCCSE search engine by exploiting the method *addPrefixes*, admitting a list of pairs, each one endowing both the prefix and the prefixed IRI.

Queries can be instantiated by exploiting the Query Builder module. Users may use one the default queries derived from the competency questions of POC4COMMERCE or define new ones. POC4COMMERCE default queries may be performed by calling a specific method for each query. There are two types of default queries, the parametric queries and the non-parametric queries. For instance, the query QE1 (see Deliverable 3) can be performed by calling the method *performQueryQE1*, possibly passing a list of pairs representing additional prefixes to be used in the query. Parametric queries can be performed in an analogous way. Additionally, they require that the query parameters are passed to the method. In order to perform the standard queries, the following method are provided:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

- *performQueryQF1([prefix, IRI])*, performs the query QF1 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query.
- *performQueryQF2([prefix, IRI])*, performs the query QF1 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query.
- *performQueryQF3([prefix, IRI], parameter)*, performs the query QF3 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQF4([prefix, IRI], parameter])*,performs the query QF4 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQF5([prefix, IRI], parameter)*, performs the query QF5 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQF6([prefix, IRI], parameter)*, performs the query QF6 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQF7([prefix, IRI], parameter)*, performs the query QF7 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQC1([prefix, IRI])*, performs the query QC1 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query.
- *performQueryQC2([prefix, IRI], parameter)*, performs the query QC2 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.

  performQueryQC3([prefix, IRI]), performs the query QC3 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query.
- *performQueryQE1([prefix, IRI])*, performs the query QE1 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

- *performQueryQE2([prefix, IRI], parameter)*, performs the query QE2 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQE3([prefix, IRI], parameter)*, performs the query QE3 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query, whereas *parameter* the parameter to be passed to the query.
- *performQueryQE4([prefix, IRI])*, performs the query QF3 and returns the result as JSON, where *prefix* and *IRI* represent the prefix and the IRI to use in the query.

Finally, users may define their own custom queries. Custom queries are defined by instantiating the class *Query* and then calling the OCCSE method *performQuery*, passing the created query. Queries admit three parameters:

- A list of pairs, each one representing the prefix and the prefixed IRI to be used in the query.
- A string representing the query to be performed.
- A list of pairs, each one representing the variable to be replaced inside the parametric query, and the parameter.

Results of queries are returned in JSON format in the form of:

$$\begin{array}{l} [\\ \ [\, r_1,\ [\, [v_1, b_1^1],\ \ldots,\ [v_s, b_1^s]\,], \\ \ \ldots, \\ \ [\, r_n, [\, [v_1, b_n^1],\ \ldots,\ [v_s, b_n^s]\,] \\ ] \end{array}$$

where

- $r_1, \ldots, r_n$ represent identification codes for the results $i, \ldots, n$, respectively
- $v_1, \ldots v_s$ represent the query variables $i, \ldots, s$, respectively
- the element b_i^j represents the binding for the variable j of the result i, for $j = \{1, \ldots, s\}$ and $i = \{1, \ldots, n\}$.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

1.4 RestAPIs for this service

Currently, no REST services are foreseen for the current implementation.

1.5 Ontologies

The POC4COMMERCE ontological stack is explained in details in the Deliverable 3. In Table 3 a brief summary is reported.

Name	Description	Domain	Originality	Language
OC-Found	Semantic descriptions of the stakeholders of the ONTOCHAIN ecosystem, including supply chains of resources, digital identities of agents, quality valuation processes	ONTOCHAIN ecosystem	New but adopting the GPL2 ontology OASIS	OWL 2
OC-Commerce	Semantic descriptions of offerings, auctions and commercial activities	e-commerce	New but adopting the OC-Found ontology and the Creative Commons Attribution 3.0 ontology GoodRelations [4]	OWL 2
OC-Ethereum	Semantic descriptions of Ethereum blockchain, smart contracts, and tokens	Ethereum blockchain	New but adopting the ontology OC-Commerce and the GPL2 ontology BLOONDiE [5]	OWL 2

Table 3: Summary of the POC4COMMERCE ontological stack

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

1.5.1 The iExec Case Study

In this section, we show how the POC4COMMERCE ontological stack is applied to a real world use case of the iExec network. Section 1.5.1.1 provides an overview of the basic concepts and functioning of the iExec network, then Section 1.5.1.2 illustrates in detail its ontological representation.

1.5.1.1 Brokering in iExec

The *iExec network* connects sellers with buyers of cloud resources,[4]. Specifically, cloud resources are of the following three main types:

- applications,
- datasets, and
- computing resources.

Applications are standalone computer programs that can be downloaded and executed, eventually providing execution parameters and input files, by a remote machine as a *task*. A task execution will produce a file containing the results of the computation.

Applications can process data provided in *datasets* available on the iExec network. Finally, computational resources required to carry out application executions are provided by *workers*, i.e., machines on the iExec network which are capable to download and execute applications (in the meaning of iExec).

Application providers, namely, actors providing applications via the iExec network, can define commercial conditions (in particular, usage fees) for application executions. Such commercial conditions are encoded into the so called *app orders* which, in their turns, are published on the *iExec Market Place*.

The structure of app orders is described in Figure 2, where

- `app` is a unique identifier for the application;[5]
- `appprice` is the price for a single execution of the app;
- `volume` is the maximum number of app executions related to the order;

[4]`https://docs.iex.ec/`

[5]Specifically, app is the address of a smart contract associated with the application.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

```
struct AppOrder
{
  address app;
  uint256 appprice;
  uint256 volume;
  uint256 tag;
  address datasetrestrict;
  address workerpoolrestrict;
  address requesterrestrict;
  bytes32 salt;
  bytes   sign;
}
```

Figure 2: App order structure

- `tags` are application-specific additional computational requirements (for example, execution in a trusted environment);
- `datasetrestrict` and `workerpoolrestrict`, when specified, restrict the executions related to the order to a specific dataset and/or to the specified worker pool;
- `requesterrestrict`, when specified, provides additional restrictions of execution requests, which will be described later, related to the order;
- `salt` is a random value to ensure order uniqueness;
- `sign` is the cryptographic signature, using the EIP712[6] structure signature mechanism, of the order.

In analogous way, *dataset providers*, namely, actors providing datasets via the iExec network, publish on the iExex marketplace *dataset orders* describing the commercial conditions regarding the datasets to be used in task executions. The structure of dataset orders, reported in Figure 3, is very close to the structure of app order described above, hence the semantics of the dataset order fields can be easily deduced from the analogous app order fields.

[6]`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md`

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

```
struct DatasetOrder
{
  address dataset;
  uint256 datasetprice;
  uint256 volume;
  uint256 tag;
  address apprestrict;
  address workerpoolrestrict;
  address requesterrestrict;
  bytes32 salt;
  bytes   sign;
}
```

Figure 3: Dataset order structure

Workers are grouped into *worker pools*, and each worker pool has a *worker pool manager*. Any application execution is performed by a worker pool following the so called *Proof of Contribution protocol*, in short *PoCo*, described in [1, 2]. During this phase, workers can eventually retrieve the dataset required for the execution. Each execution is started by the worker pool manager, which acts as *scheduler* during the corresponding *PoCo* protocol run. Further details about how worker pools perform application executions are out of the scope of this document.

Commercial conditions about the usage of computational resources in the worker machines of the worker pool are defined and published by the worker pool manager on the iExec market place *workerpool orders*, described in Figure 4. Again, the fields of the latter structure are similar to those reported for app orders, except for two fields with no correspondents in app orders:

- `category` which describe, in some sense, the *size* of the computation in terms of maximum task execution time (see Table 4);[7]
- `trust` is a confidence level for accepting contributions of workers in the PoCo execution.[8]

Users who need to perform computation are called *requester*. They can retrieve app orders, worker pool orders and dataset orders from the iExec market place or by other means. However, such orders

[7] `https://docs.iex.ec/key-concepts/pay-per-task-model`
[8] `https://medium.com/iex-ec/`

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

```
struct WorkerpoolOrder
{
  address workerpool;
  uint256 workerpoolprice;
  uint256 volume;
  uint256 tag;
  uint256 category;
  uint256 trust;
  address apprestrict;
  address datasetrestrict;
  address requesterrestrict;
  bytes32 salt;
  bytes   sign;
}
```

Figure 4: Workerpool order structure

Category	Maximum Elapsed Time
XS	5 min
S	20 min
M	1 hour
L	3 hour
XL	10 hour

Table 4: Categories

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

```
struct RequestOrder
{
  address app;
  uint256 appmaxprice;
  address dataset;
  uint256 datasetmaxprice;
  address workerpool;
  uint256 workerpoolmaxprice;
  address requester;
  uint256 volume;
  uint256 tag;
  uint256 category;
  uint256 trust;
  address beneficiary;
  address callback;
  string  params;
  bytes32 salt;
  bytes   sign;
}
```

Figure 5: Request order structure

are signed by their providers, so that they can be eventually used in disputes. Once the requester has acquired a suitable app order, a suitable workerpool order and, eventually, a suitable dataset order, he have to create a *request order* matching all the restrictions in the provider orders. As for orders of all other types, such a request order, whose structure is described in Figure 5, has to be signed by the requester. Then, the selected orders among with the request order can be sent to the *iExec clerk* smart contract. The iExec clerk verifies the orders' signatures and compatibility and, if signatures are valid and all the required conditions are met, writes the agreement on the blockchain. Then, PoCo protocol is started in order to perform the execution. The just mentioned process of requesting an execution is reported in detail in [3].

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

1.5.1.2 Representing iExec orders in POC4COMMERCE

Having summarized brokering and provisioning of executions on the iExec network, this section outlines a way to encode the structures described in Section 1.5.1.1 in the POC4COMMERCE ontological stack.

The proposed encoding focuses on offerings of assets provided through the iExec network, in order to ease the discovery of services and commercial conditions provided via this network.

Novel classes, properties and individuals that describe notions specific of the iExec network are introduced. For those iExec specifc elements, which however fit in the POC4COMMERCE ontological stack, we will use the namespace

iexec : `http://ontology.iex.ec`

Let us first analyze the items traded through the iExec network. Essentially, they are *executions* of iExec applications. As recalled before, executions are first characterized by:

- the application that will be executed,
- the worker pool, with the corresponding worker pool manager, which will have in charge the execution, and, eventually,
- a dataset against which execute the application.

Applications and datasets are *assets* whose utilization in execution can be bought. So, they can be represented as instances of two subclasses of `oasis:DigitalServiceAsset`: `iexec:Application` and `iexec:Dataset`, respectively.

Describing applications and datasets in our modelling is not essential for our purposes, except for asset identifiers, which may be encoded in individual IRIs by defining specific naming schemes. However, applications and dataset modelling could be extended, where needed, with other information, such as, for example, application and dataset provider, which should be modelled as `oasis:Agent` instances.

As recalled before, executions are actually performed by *worker pools*. They are organizations of agents, coordinated by worker pool managers. For worker pools and worker pool managers, we introduce the classes `iexec:WorkerPool` and `iexec:WorkerPoolManager`. Each worker pool is connected to its manager via the property `iexec:hasWorkerPoolManager`.

As it can be seen from the structure of request orders reported in Figure 5, executions also have optional fields:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

- `tag`, represented by the property `iexec:hasTag`, is left unspecified as it is a specific feature provided by the application or by the worker pool, so that application providers and worker pools can provide ad-hoc taxonomies of tags;
- `category`, which indicates the maximum elapsed time for the execution, can be specified in the execution by means of the property `iexec:hasCategory`, having ad range the class corresponding to the categories in Table 4.
- `trust`, which is an integer value indicating a confidence level for the computation result, can be specified for an execution via the `iexec:hasTrust` datatype property;
- `params`, which are documents publicly available on the web, are used in the execution as input parameters for the application and can be specified by means of the property `iexec:hasParam`.

We denote the group of properties defined above as *optional execution properties*.

Now that we defined how to represent assets of interest with our ontological stack, we focus our attention on publishing *offerings* to sell these. For `AppOrder`, described in Figure 2, `DatasetOrder`, described in Figure 3, and `WorkerPoolOrder`, described in Figure 4, we create three corresponding subclasses of `Offering` (defined in OC-Commerce): `iexex:AppOffering`, `iexec:DatasetOffering`, and `iexec:WorkerPoolOffering`. Converting iexec orders into corresponding POC4COMMERCE compliant individuals is really intuitive:

- the order identifier must be included in the offering individual IRI, given a IRI scheme provided for this purpose;
- prices are provided as instances of the class `UnitPriceSpecification`, directly imported in OC-Commerce from the GoodRelations ontology;
- contents of the `volume` fields correspond to the notion of *eligible quantity* in GoodRelations;
- `apprestrict`, `datasetrestrict`, and `requestrestrict`, for the sake of simplicity, are not taken into account here, but could be easily modelled by introducing novel properties for this purpose;
- optional properties are modelled with the *optional execution properties* introduced before;
- `salt` and `sign` are out of the scope of this deliverable, and can be retrieved if needed by directly accessing the original order from which the offering was created.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Applications and Datasets are particular assets that cannot be *released* or *delivered* but can be used in the context of an execution. Thus, defining supply chains for these assets would not be appropriate. As consequence, individuals of the classes representing the corresponding offerings, i.e., `AppOffering` and `DatasetOffering`, are not associated with supply chains.

Conversely, the execution of tasks is actually performed by the worker pool agent, whose behaviour is depicted in Figure 7. The agent accepts as input the iExec deal and computes the corresponding iExec task. Subsequently, the iExec clerk smart contract verifies a request order and signs up a corresponding *deal*, then stores it on the blockchain. This process can be modelled in the supply chains of the `iexec:WorkerPoolOffering` described in Figure 6. For this purpose, we define the class `iexec:RequestOrder` to represent the `RequestOrder` as described in Figure 5. The conversion of a `RequestOrder` object into a corresponding `iexec:RequestOrder` individual works similarly to order objects, except for `beneficiary` and `callback`.

The class `iexec:RequestOrder` is introduced to model the iExec clerk agent as described in Figure 8. For this agent, we define a suitable behaviour, namely `iexec:establishDeal`, admitting the following parameters:

- an Application offering (corresponding to an app order),
- optionally, a Dataset Offering (corresponding to a Dataset order),
- a WorkerPool offering (corresponding to an WorkerPool order)
- a `iexec:RequestOrder` representing the execution request.

The iExec clerk behaviour consists of the action `iexec:validate`, which checks that the set of orders corresponding to parameters is valid, and which produces the deal as output.

The `ProofOfWork` supply chain of the workerpool order is related with the behaviour of the iExec clerk agent (see Figure 8), whereas the `Release` supply chain consists of an activity exploiting the behaviour of the worker pool agent (Figure 7).

Finally, the `Payment` supply chain consists of an activity implementing the payment using the iExec digital currency *RLC* manager by the *iEx.ec_Network_Token* smart contract.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

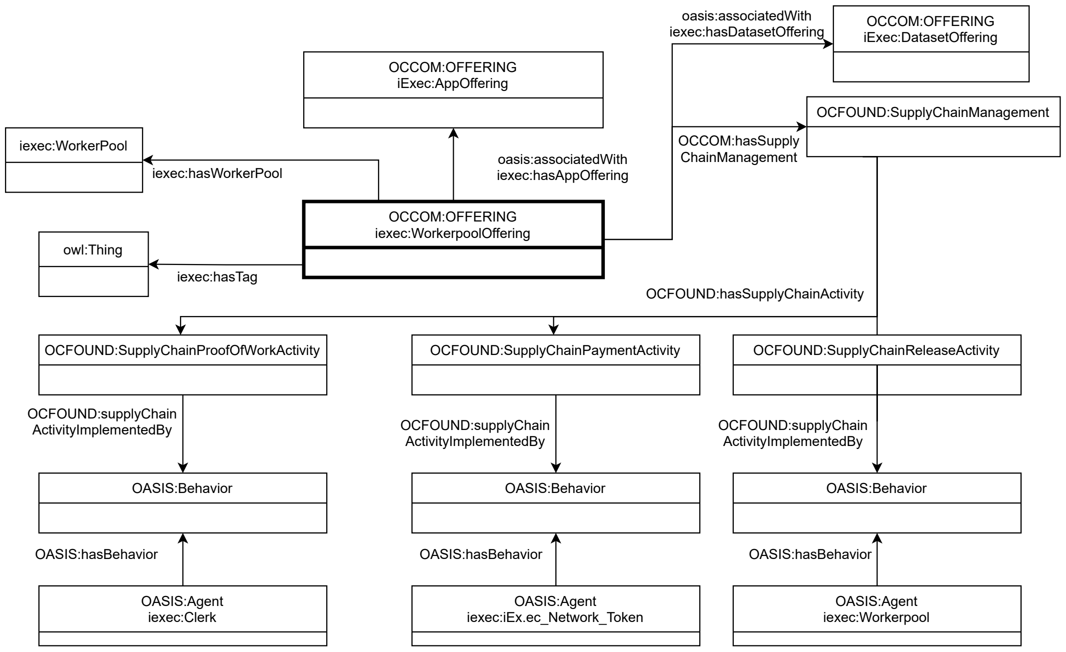

Figure 6: Workerpool offering

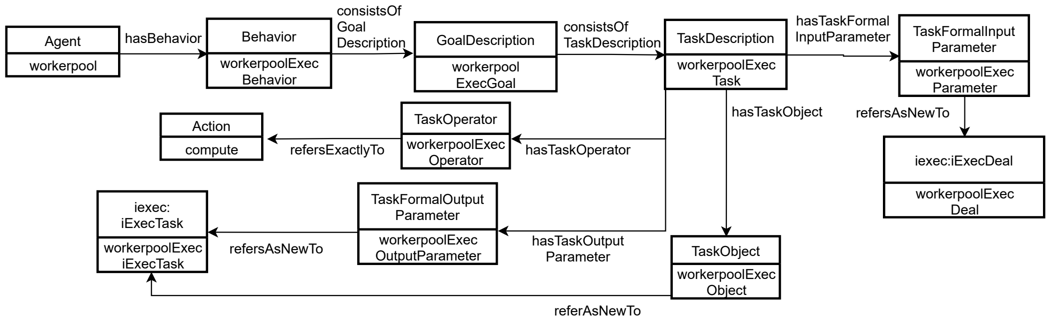

Figure 7: Workerpool agent

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

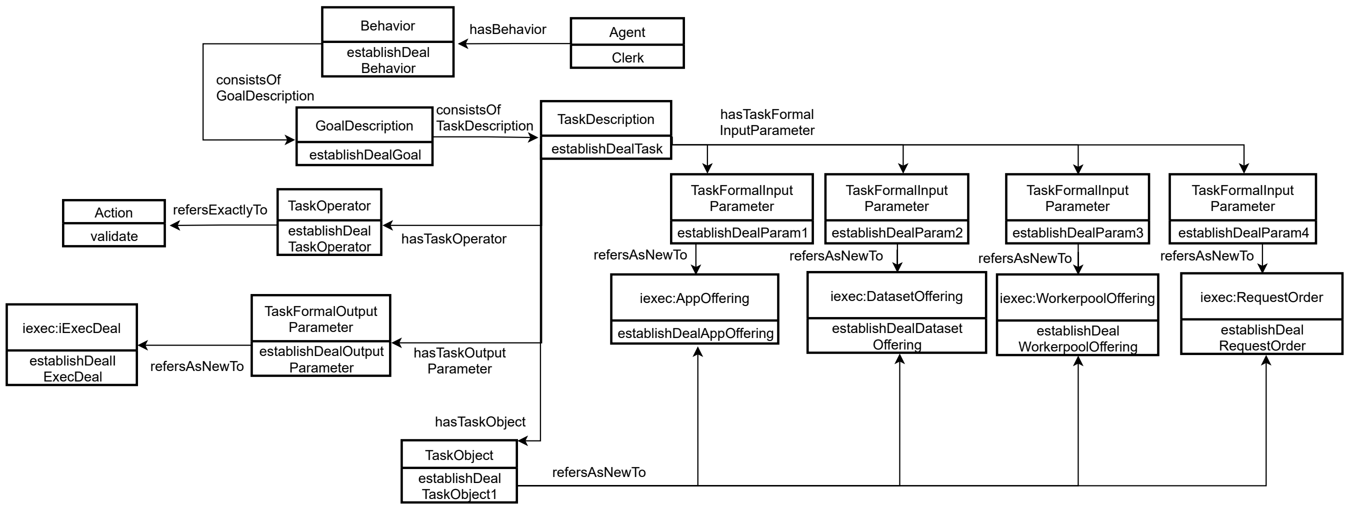

Figure 8: iExec clerk agent

1.6 How to compile, run, deploy, and test the code

In order to work with the two prototype modules, the Python interpreter version 3.7 or greater is required. Moreover, the interpreter must be provided with, at least, the RDFLib library version 5.0.0 for the OCGEN module, and with the library OWLReady 2 version 0.33 for the OCCSE module. It is strongly suggested to also install the Cython parser module for better performances. To compile and run the code it is sufficient to create a new Python program using the APIs provided by either the OCGEN module or by the OCCSE module and to launch it with Python. The deploy of the application can be carried on as any other Python program. A test example for the OCGEN module is available in the repository folder *python/test/ocgen-test/ocgen-test.py*, whereas a test example for the OCCSE module is available at *test/occse-test/test_occse.py*. The repository is also provided with a *Readme* file explaining in details how to use the APIs.

2 Technical Value Added

POC4COMMERCE contributes to many aspects of the ONTOCHAIN ecosystem. From one hand the ontological layers build the foundations of the ONTOCHAIN ecosystem thus making it readily functioning to build a sustainable, interoperable, and trustworthy eCommerce environment for people

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

and software to interoperate. We recall that the foundational ontology OC-Found provides a unifying canvas for all ONTOCHAIN participants and relationships between them, a substrate describing and connecting what really exists in ONTOCHAIN, with the relevant stakeholders interoperating coherently in a heterogeneous context. OC-Found also lays the grounds for the semantic interoperability of potentially innumerable domain-specific ontologies for the ONTOCHAIN ecosystem, such as those for eScience, eEducation, eHealth, eGovernment, eCommerce, eTourism, and eInfrastructures. OC-Commerce impacts on how digital commerce is carried out in particular within the blockchain environment, thus realizing an affordable marketplaces where sellers and buyers may freely choose the services associated with their business activities. OC-Ethereum advances the concept of *semantic blockchain*. A semantic blockchain implies that smart contracts can be referenced without pre-existing knowledge of their deployment and of the underlying programming code: their functionalities are fully specified by formal and machine-understandable representations, thus realizing an interoperable environment where off-chain services interact with applications lying on many blockchains such as Ethereum, Cardano and Stellar Lumens. On the other hand, the semantic approach pursued by POC4COMMERCE culminates in the OCCSe search engine allowing people and software to practically probe the semantic data, finding offerings, agents, and desiderata, thus enabling the semantic blockchain conceptualized by the ontological stack. Moreover, the OCGEN module enables users to generate their own data thus integrating blockchain information with off-chain data, store both locally and on remote services through *federation* processes. Finally, semantic annoted blockchains enables integration with other blockchain moving forward the realization of cross-chain distributed applications.

2.1 Key Innovations of your solution, in summary

POC4COMMERCE innovates the ontological representation of blockchain-oriented digital commerce by integrating and extending the most promising ontologies in the field by providing a semantic descriptions of smart contracts and related transactions, in particular of smart contracts related with tokens trading and associated with commercial means: all these ontologies are conjoined and extended to also cover the gap missing from the literature on the representation of digital tokens, smart contracts, digital identities, valuation mechanisms, and auctions. The POC4COMMERCE vision culminates in the OCCSE search engine that practically enabling users and applications to probe the blockchains and conjoining data stored in transactions with out-chain data.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

2.2 Eventual suggested evolutions of your solution

In order to being part of the human-centric evolution of the internet, both POC4COMMERCE ontological stack and search engine enjoins some new features. Most notably,

- the OCGEN module should be integrated by smart contract programming languages in order to be embedded within transactions and included as part of the transactions verification process: automatically generated semantic annotations for smart contracts in the shape of OC-Ethereum ontology concretely enables interlinking of blockchains and cross-chain dApps.
- the OCCSE module, currently running on the client side, should be run on specific suitable blockchain nodes in order to allow IoT devices and Edge Artificial Intelligence to leverage blockchain dApps.
- Suitable editors provided with GUIs interfaces should assist smart contract developers to generate smart contracts behaviors in the shape of the OC-Found ontology and to annotate transaction with RDF graphs derived from OC-Ethereum.
- A pletora of applications, even out of the blockchain, can be also implemented by exploiting the epistemological vision of the OC-Commerce ontology thus delivering Web 3.0 trustworthy commercial services.

2.3 Present or future patentability of your solution

Concerning the ontological stack, the epidemiological choices introduced by the ontologies OC-Found and OC-Ethereum can be subjected to the patentability process, where OC-Commerce not, due to the licence limitations of GoodRelations. The search engine adopts standard open sources libraries such as RDFLib, OWLReady 2, and the reasoners PelleT and HermiT. Hence, any application exploiting the OCCSE engine can be patented.

2.4 Technical KPIs

In this section we illustrates the KPIs for the POC4COMMERCE prototype that extends the once presented in Deliverable 3.

2.4.1 Interoperability and standardization

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

KPI	Can apply to your project	Did you implement it?	Summarize your contribution
Interledger APIs	Yes		
Allow for different consensus protocols and smart contract implementations	Yes	Yes	The OC-Ethereum ontology permits the definition of any new generation of smart contracts that semantically annotate the generated transactions; the current version features fungible, non-fungible and semi-fungible tokens
Did you propose or could/will propose standards/ drafts?	Yes	No	The ontology OC-Found and OC-Ethereum can be interpreted as drafts to develop further as well major milestones towards standardisation of semantic blockchains

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Describe international events on standardization activities participated/ contributed	Yes	No	The POC4COMMERCE ontological stack can be further extended towards possible standardisation activities

2.4.2 KPIs towards innovation

KPI	Can apply to your project	Did you implement?	Summarize your contribution
Did you implement new innovative ONTOCHAIN use cases?	Yes	Yes	A relevant use case is that of an apple vendor adopting the semantic blockchain as a means to reach potential customers who may conveniently utilise the semantic search engine
Did you implement new innovative ONTOCHAIN reasoning technologies?	No		
Did you implement new ways to serialise ontologies and semantics on blockchain?	No		

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Did you implement other new approaches to implementing "immutable" semantics and reasoning?	No		

2.4.3 KPIs towards more human-centric evolution of the internet

KPI	Can apply to your project	Did you implement?	Summarize your contribution
Which is the Trust Assessment Effectiveness, e.g., accuracy for subjects or for content, for your solution?	Yes	Yes	The ontological approach of POC4COMMERCE, in combination with the use of semantic web reasoners, then enabled by the semantic search engine, may contribute to any agent's trust on subjects or contents
How can you assess the privacy/anonymity of your solution?	No		
Did you implement of zero knowledge proof protocols?	No		
Details the size of decentralized storage, in GB	No		

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Detail the number of applications deployed	Yes	Yes	Two modules are deployed, namely OCGEN for generating agent data and OCCSE for querying and reasoning on the semantic knowledge base

2.4.4 KPIs towards more decentralized NGI

KPI	Can apply to your project	Did you implement?	Summarize your contribution
Did you implement new decentralised computing technologies for storing and accessing data, e.g., via the OAI- PMH protocol, that achieve high reliability, availability, Quality of Service, and similar properties necessary to realise new decentralised services?	No		
Did you implement new decentralised social networks?	No		
Did you implement new decentralised publishing platforms?	No		

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Did you implement new Digital Twin technologies that can help establish digital representation of the reality in specific circumstances where needed?	No		

2.4.5 KPIs towards new forms of interaction and immersive environments for NGI users

KPI	Can apply to your project	Did you implement?	Summarize your contribution
Did you implement new human to Internet interaction paradigms developed in the use cases of ONTOCHAIN?	Yes	Yes	The POC4COMMERCE ontological stack and the semantic search engine are exposed through APIs that support human understanding and discernement about the blockchain and also digital commerce as carried out on the WEB 3.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Did you implement decentralized apps in ONTOCHAIN that involve human interactions in education, energy, finance, governance, healthcare, identity, interoperability, mobility, privacy, public sector, real estate, social impact, supply chain?	No		

2.4.6 KPIs related to the implementation

KPI	Can apply to your project	Did you implement?	Summarize your contribution
Code simplicity (analyser used and results)	Yes	No	As Python applications, the OCCSE and OCGEN modules can be subject to software code analysis in the future. However, ontological metrics were successfully applied to the ontological stack (see Deliverable 3)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Testability Coverage	Yes	No	As Python applications, the OCCSE and OC-GEN modules are well amendable to standard testability coverage as a possible future activity

3 Communication, Dissemination, Exploitation

3.1 Your contents

3.1.1 Main achievements

POC4COMMERCE achieves a core ontological representation of WEB 3.0 commerce carried out over Ethereum. It provides an ecosystem of modular Semantic Web ontologies describing the essential semantic compartments of blockchain-based electronic commerce.

Leveraging the blockchain for electronic commerce is a recent business opportunity revolving around the dynamics and exchange mechanisms of blockchain tokens. However, the experience of querying the blockchain for identifying a desired token with its associated features, or the smart contract that trades it at what conditions may not turn out practical due to present the lack of suitable technologies.

This is the main problem that POC4COMMERCE addresses. The semantic descriptions of Ethereum smart contracts and related transactions, notably of the smart contracts related to token trading, allow us to port semantic-query languages such as SPARQL to the blockchain domain. This entails a more usable and aware interaction with the blockchain, ultimately enhancing its widespread application and layman's acceptance.

POC4COMMERCE reuses, adapts, extends, and puts into practice the best available technologies for semantic representation to build a novel, three-level stack of ontologies. More precisely, the project builds that stack by leveraging building blocks such as the **OASIS** ontology for agents, the **BLONDiE** ontology for the Ethereum blockchain, and the **GoodRelations** ontology for commercial offerings are leveraged. The stack rests on the OC-Found ontology modelling ONTOCHAIN

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

participants and actors, then continues with the OC-Commerce ontology modelling commercial offers, products and services, and culminates with the OC-Ethereum ontology modelling the Ethereum blockchain, smart contracts and digital tokens exchanged for commercial purposes.

POC4COMMERCE also delivers effective tools to access the rich and diverse knowledge base that the ontological stack makes available. The main tool is the OC-Commerce Search Engine (in short, OC-CSE, see `https://github.com/ONTOCHAIN/POC4COMMERCE`) to profitably find goods, products, information, and services published in the ONTOCHAIN digital market. OC-CSE exposes an API and hence may be profitably called by any software agent in the ONTOCHAIN ecosystem. The engine is constructed by means of a combination of Semantic Web tools, reasoning services, and SPARQL queries.

POC4COMMERCE exploits the knowledge representation and reasoning capabilities of web ontologies to practically realize the semantic core of ONTOCHAIN and how the ONTOCHAIN ecosystem shapes up in the eCommerce vertical domain.

Finally, *Semantic Blockchain* must be recalled. It is a vision of blockchains enabled with semantically annotated transactions. It contributes to various purposes, such as the interlinking with out-of chain information and the automatic discovery of smart contracts.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

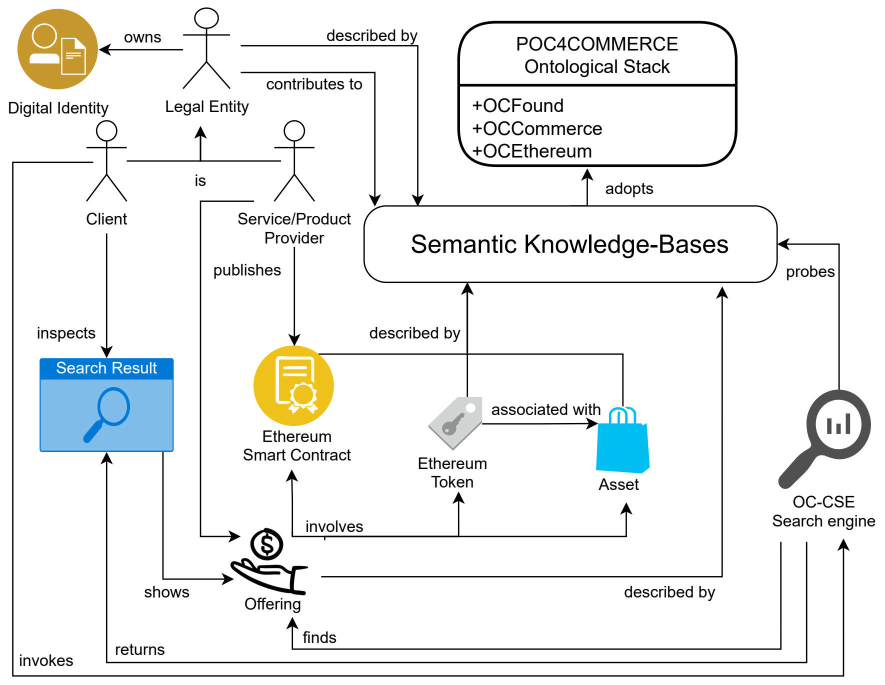

Figure 9: POC4COMMERCE vision of the Semantic Blockchain

Figure 9 provides a graphical representation of how POC4COMMERCE and its objectives concur to the mission of the Semantic Blockchain.

3.1.2 Demo

The demo is available at `https://www.youtube.com/watch?v=MuY3slZwN4A`. The audio transcription of the demo is the following.

Slide 1. Welcome to the Demo for POC4COMMERCE. Let's start by explaining what the POC4COMMERCE project is about.

Slide 2. The goal of POC4COMMERCE is to provide a consistent, unambiguous, and shared semantic model for the ONTOCHAIN ecosystem. POC4COMMERCE promotes an epistemological process delivering both an ontological stack for the building blocks of ONTOCHAIN in general and for the commercial domain in particular. The project also delivers APIs for generating, reasoning on, and

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

querying ontological data.

Slide 3. The ontological stack of POC4COMMERCE provides consistent and practical means for describing any participant of the ONTOCHAIN ecosystem. The ontologies also describe actors, in particular commercial actors, physical and digital assets, agent interactions, offerings on assets, price determination mechanisms, supply chains, smart contracts, and tokens. On top of the ontological stack POC4COMMERCE delivers suitable APIs that enable users to generate their own data, to query and reason on them. Moreover, users adopt the APIs to conveniently find services, products and information.

Slide 4. The ontological stack consists of 3 main ontologies:

- The OC-Found ontology, that extends the OASIS ontology, describes agents in terms of their behaviors and the actions associated with the behaviors that generate those actions. Also called plans, agent actions represent the willingness of agents to activate some behaviors.
- The OC-Commerce ontology, that extends the OC-Found and GoodRelations ontologies, describes commercial stakeholders, including offerings and auctions.
- The OC-Ethereum ontology, which extends the OC-Commerce ontology and the Blondie ontology, describes the Ethereum blockchain, related smart contracts, and tokens.

Slide 5. Let's move on to the APIs that POC4COMMERCE implements for generating, reasoning on, and querying data. Specifically, POC4COMMERCE provides two main modules.

- The first module, called OCGEN, is a Python monolithic module exploiting the RDFLib library. It provides APIs that enable users to generate their one data, in particular agent behaviors.
- The second module, called OC-Commerce Search Engine, provides APIs to load ontological data, to reason on them, and to perform SPARQL queries. Specifically, the module permits the definition of user-defined queries and includes also a set of predefined POC4COMMERCE queries implementing suitable competency questions.

Demo: OCGEN (showing APIs)

Let's see how to use the APIs provided by the OCGEN module. First, we create the OCGEN object by passing three RDF graphs:

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

- The RDF graph storing the agent behaviors.
- Possibly the RDF graph storing the behavior templates.
- The RDF graph storing offerings, blockchain transactions, and all the user data.

Just for testing, we adopt the same RDF graph for all the data and we store it locally in the file called ocgen-test.owl.

We can now define our agents. We first call the function createAgent using the name of agent as parameter. We can also use the functions *addObjPropAssertion* to specify some object property assertions about our agent or the method *addClassAssertion* to include some class assertions. For example, we can specify the digital identity of our agent.

We now assign some behaviors to our agent. We call the function *createAgentBehavior* specifying the name of the behavior, the name of the goal and of the task. Then we add the operator, possibly the operator arguments, the task object, input and output parameters. Finally, once we have defined all the behaviors, we can connect them to our agent by calling the method *connectAgentToBehavior*, specifying the agent name and the behavior.

In analogous way, we can also create agent templates and connect them to the concrete agent behaviors. Once our RDF graph is complete, we serialize and share our data by writing the RDF graph on a file . Once data are available, we can start the Search Engine module and make some queries.
Slide 6.

The Search Engine module is constituted by six parts.

- A repository builder collecting and storing in the local Triple Store data selected by users in order to perform reasoning and querying.
- A reasoner interface, allowing users to choose the preferred reasoner such as HermiT and Pellet.
- A query builder, enabling users to define their own SPARQL queries.
- A reasoner and a SPARQL query engine.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

Demo: OCCSE (showing APIs)

To use the Search Engine, we need to initialize some modules. First, we need a repository manager that allows us to collect RDF graphs. We use the method *addRepositories* indicating the URL of the RDF graphs. For example we load the entire POC4COMMERCE ontological stack and the data create with the OCGEN module.

Now we need to specify the reasoner we want to use. We create the ReasonerInterface module passing the name of the reasoner, in this case the Pellet reasoner.
To instantiate the search engine we create the OCCSE object passing the repository manager and the reasoner interface.
We can call the method *loadRepository* to load all the data and the method *syncReasoner* to start reasoning with the reasoner selected before.
We are now ready to perform query. We can either run the POC4COMMERCE standard queries or create some custom queries.
To perform POC4COMMERCE standard queries, we call the method performQuery followed by the name of the query, for example QF1. Queries require the prefixes used for the query. There also parametric queries that additionally require to specify the parameter introduced, for example for the query QF3, the parameter is the name of the offering.
User-defined queries work same way. We first create a Query object by specifying the prefix used, and the SPARQL query we want to perform. We just need to call the method *performQuery* and passing the created query to see the output.
We can see the execution of the reasoner and the output of the queries. The output is serialized in the JSON format: it is constituted by a set of results. Each result contains the variables introduced in the query and the corresponding matches.
Slide 7: That's all, thank you for your attention.

3.1.3 Your testimonial

The ONTOCHAIN adventure has been thrilling and invigorating. We were lucky since the beginning to find a call for project proposals that was the perfect cut for the very research lines some of us had long wanted to pursue. Good luck continued through the formation of a lean yet fit consortium to effectively pursue that research. But the best luck was the engagement in a profitable and thought-provoking mentoring process. The mentors offered continuous live review of our ideas and developments, while bolstering the business potential in a way we would have been unable ourselves,

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

ultimately navigating us to success.

3.2 Communication, dissemination, exploitation results

3.2.1 News

None, due to delays in the signing process of the grant agreement.

3.2.2 Press releases

None, due to delays in the signing process of the grant agreement.

3.2.3 Scientific Papers

The ontological approach of POC4COMMERCE has been described in the following two proceedings:

- 18th International Conference on Economics of Grids, Clouds, Systems & Services (http://2021.gecon-conference.org), 21-23, September 2021, Online, Springer LNCS Proceedings. The title of the paper is **Semantic Representation as a Key Enabler for Blockchain-Based Commerce**, by Giampaolo Bella, Domenico Cantone, Cristiano Longo, Marianna Nicolosi Asmundo and Daniele Francesco Santamaria.

- The 14th International Symposium on Intelligent Distributed Computing, 16-18, September 2021, Online Conference, Italy. The title of the paper is **Blockchains through ontologies: the case study of the Ethereum ERC721 standard in OASIS**, by Giampaolo Bella, Domenico Cantone, Cristiano Longo, Marianna Nicolosi Asmundo and Daniele Francesco Santamaria.

A paper describing the POC4COMMERCE approach will be submitted before October 25th at the Semantic Web Journal (IOS Press).

3.2.4 Exhibitions

The POC4COMMERCE project has been presented at the **European Blockchain Week: Blockchain and AI for European Green Deal, 20-24, September, Ljubljana, Slovenia (OnLine).**

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

4 Appendix: Final design updates

The design of final implementation is completely described by the deliverable D3.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338

References

[1] H. Croubois. Poco series #1 - about trust and agents incentives, 2017.

[2] H. Croubois. Poco series #3 - protocol update, 2018.

[3] H. Croubois. Poco series #5 - open decentralized brokering on the iexec platform, 2018.

[4] M. Hepp. Goodrelations: An ontology for describing products and services offers on the web. In Aldo Gangemi and Jérôme Euzenat, editors, *EKAW*, volume 5268 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2008.

[5] H. Ugarte-Rojas and B. Chullo-Llave. Blondie: Blockchain ontology with dynamic extensibility. *CoRR*, abs/2008.09518, 2020.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957338