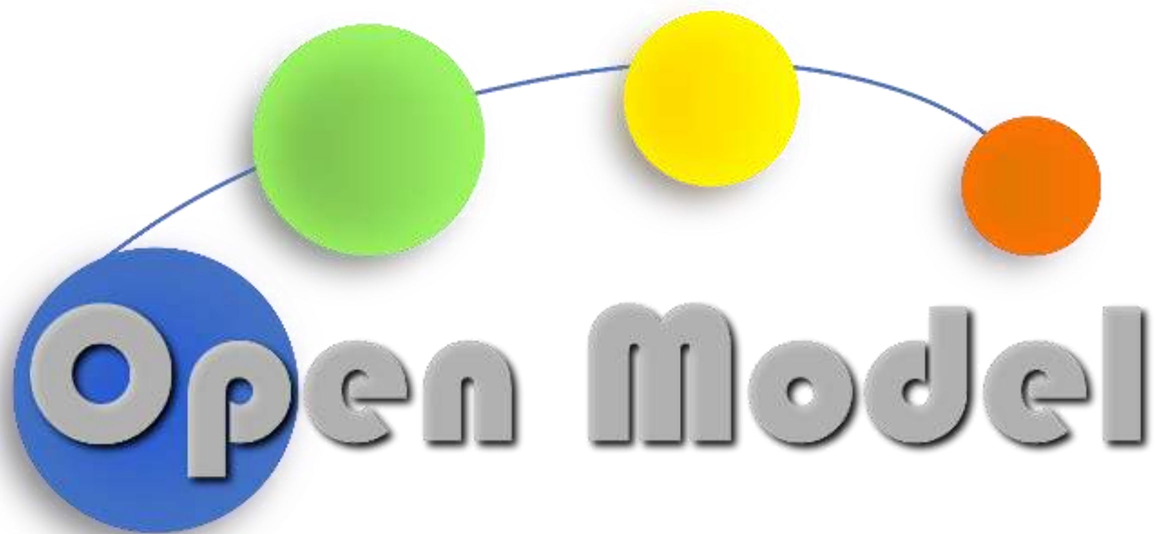
Ref. Ares(2023)713304 - 31/01/2023

2023




ALESSIO MORA (UNIBO)
LUCA FOSCHINI (UNIBO)
ALESSANDRO CALVIO (UNIBO)

# D4.2 - ONTOFLOW MVP DEPLOYMENT WITH INITIAL MCO

## DOCUMENT CONTROL

| | |
|---|---|
| Document Type | Other |
| Status | Final |
| Version | 1.0 |
| Responsible | Luca Foschini (UNIBO) |
| Author(s) | Alessio Mora (UNIBO), Alessandro Calvio (UNIBO), Luca Foschini (UNIBO) |
| Release Date | 2023-01-31 |

## ABSTRACT

This deliverable report presents the activities carried out in WP4 in order to develop the component named OntoFlow. In particular, this document focuses on the architecture of OntoFlow, the entity which manages the knowledge base and searches for all the possible mapping routes inside the semantic data. A mapping route is a chain of ordinate steps or relations that connect one concept in to another and provide a link between the input concepts and the desired output. In this document, we mainly focus on the elements that compose OntoFlow, their structure and interactions. For the sake of clarity, the document will also contextualize the proposed architecture by giving some insights from a practical point of view.

## CHANGE HISTORY

| Version | Date | Comment |
|---|---|---|
| 0.1 | 2023-01-15 | First Draft |
| 0.2 | 2023-01-30 | Described workflow generation and added more details in sec. 4 |
| 1.0 | 2023-01-30 | Final / Reviewed by Technical coordinator |

## DISSEMINATION LEVEL

| | | |
|---|---|---|
| PU | Public | X |
| PP | Restricted to other programme participants (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium (including the Commission Services) | |
| CO | Confidential, only for members of the consortium (including the Commission Services) | |

## TABLE OF CONTENT

Document Control ........................................................ 1

Abstract ........................................................ 1

Change History ........................................................ 1

Dissemination level ........................................................ 1

Table of Content ........................................................ 2

1 Introduction ........................................................ 3

1.1 Overview ........................................................ 3

1.2 Deviation From Annex 1 ........................................................ 3

2 OntoFlow General Architecture ........................................................ 4

3 OntoFlow Architecture: OntoFlowKB ........................................................ 5

3.1 Tripper and Stardog backend ........................................................ 6

4 OntoFlow Architecture: OntoFlowDM ........................................................ 7

5 YAML Generation ........................................................ 10

6 OntoFlow in action: Datamodel GENERATION ........................................................ 11

6.1 Preparation phase: Populate the knowledge base ........................................................ 11

6.2 Configuration phase: OntoFlowDM engine configuration ........................................................ 13

6.3 Execution phase: Selection of the best routes ........................................................ 14

6.4 Final phase: Generation of YAML file ........................................................ 15

7 Code references ........................................................ 16

Acknowledgment ........................................................ 17



© OpenModel Consortium CONFIDENTIAL

# 1 INTRODUCTION

## 1.1 OVERVIEW

The object of this accompanying document is describing the design of OntoFlow, which is a workflow designer and builder, with the help of a very first example of its implementation. OntoFlow is able to discover and suggest the most suitable set of models, tools, and operations to obtain a specific output starting from certain inputs. OntoFlow determines the best set of workflows navigating its knowledge base, matching diverse workflows accordingly to their semantics, and, in turn, understanding their relation and whether they can be used in a chain (i.e., in series). The most suitable chain of workflows can be determined by using simple cost heuristics, or via the use of Multi-Criteria Optimizers (MCO) as well as Machine Learning-based techniques.

OntoFlow includes two components, which are the knowledge base (OntoFlowKB) and the decision-making component (OntoFlowDM). OntoFlowKB is a triplestore database designed to store and manage semantic information and mapping between ontological concepts, and it is based on the efforts carried out in the related OntoTrans project. OntoFlowDM is the component that enables the workflow decision-making operations through the embedding or integration of different MCOs. To be maximally general, OntoFlowDM may expose a set of interfaces to accept for MCOs.

This document will provide an in-depth look into the architecture of OntoFlow, zooming in to give more details about the design of OntoFlowKB and OntoFlowDM, and how their synergic interactions lead to the selection of the best workflow to execute and the generation of a workflow descriptor, as a YAML file, with a first definition of the necessary steps. This document will not cover ExecFlow, the component in charge of the actual execution of workflows.

## 1.2 DEVIATION FROM ANNEX 1

Current deliverable does not have any deviations from GA Annex 1.


© OpenModel Consortium CONFIDENTIAL

# 2 ONTOFLOW GENERAL ARCHITECTURE

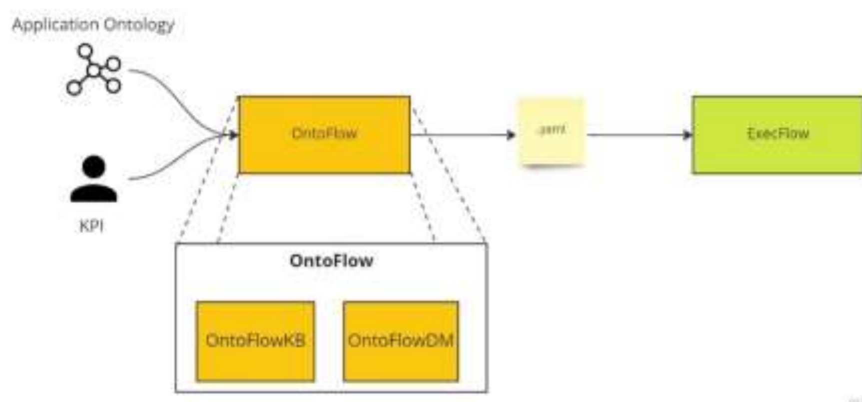



**Figure 1 - General architecture for OntoFlow and its interactions with ExecFlow**

Figure 1 depicts a first high-level overview of the OpenModel Platform (OMP) Workflow Support that handles the exploration, detection, selection, and execution of the best workflows for specific use cases and relies on the work of its two main components: OntoFlow and ExecFlow. In this context, OntoFlow is the first key component to come into play and, through its two submodules OntoFlowKB and OntoFlowDM, is responsible for storing and managing the semantic representation of the elements of interest for the individual use cases and for using decision-making and MCO techniques to choose the most suitable workflow among the possible ones.

OntoFlow usually needs two inputs: the first consists of the specific ontological representation of all models, tools, targets, and attributes related to a particular use case of interest; the second refers to the definition of Key Performance Indicators (KPIs), i.e., metrics of interest for the chosen use case used to guide the workflow optimization process. The semantic representation is stored in OntoFlowKB, integrated with existing knowledge, and used as a starting point to identify all possible workflows that can produce the desired output. Once they have been found, OntoFlowDM performs an optimization to determine the best workflow based not only on the defined KPIs but also on ontological attributes that may, in some way, influence the final output. It is noteworthy that the KPIs used as input may also conflict with each other (e.g., execution speed vs. result accuracy) and that, in this case, the search process will be geared toward finding the best trade-off between them.

At the end of this first phase, the optimum workflow is identified and needs to be executed to produce the desired result. However, it is important to point out that the workflow, at this stage, is described in purely ontological terms that cannot be used by ExecFlow as is. For this reason, an intermediate descriptive file has been designed to instrument ExecFlow. In practice, the ontological workflow is described in a declarative file that details the various steps in the workflow, the values of the input elements and how they can be retrieved, as well as how the intermediate results are linked to the next steps. ExecFlow, the component responsible for the actual workflow executions, starts from this descriptive file to build, piece by piece, a pipeline of steps in a way that is specific to the actual simulation platform (e.g., AiiDA).

The following sections will provide a more in-depth discussion of the components that constitute OntoFlow's MVP, with a focus on its implementation aspects. In the end, what is explained will be shown through a small concrete use case that also helps to understand how the various elements work from a practical point of view.


© OpenModel Consortium CONFIDENTIAL

# 3 ONTOFLOW ARCHITECTURE: ONTOFLOWKB

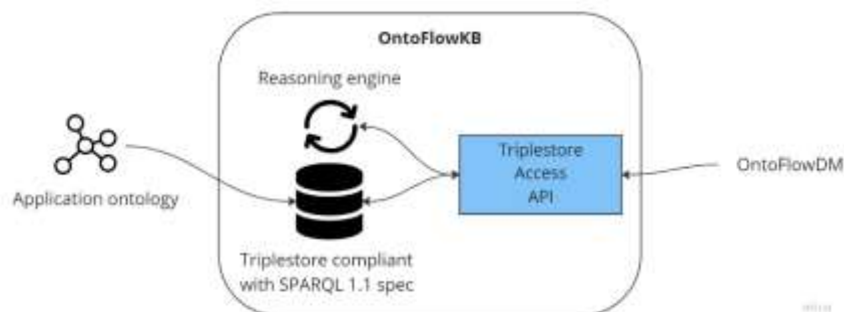



**Figure 2 - OntoFlowKB component overview**

OntoFlowKB, depicted in Figure 2, is a triplestore-based component that provides:

- Independence from the actual vendor-specific triplestore technology.
- Storing and managing of workflow-related ontological concepts (EMMO ontology developed in WP1 ) and application specific ones. The main information included concerns the definition of use case, models, tools and workflows with their attributes.
- Querying of stored ontologies by interacting with the triplestore SPARQL endpoint.
- Reasoning capabilities provided by the built-in engine able to perform reasoning query up to the OWL-DL level[1]. The ability to make inferred queries allows the identification of additional workflows beyond those directly deriving from the knowledge base.
- Information related to the mapping between ontological concepts of different application's cases.

For the sake of demonstration, the current OntoFlow MVP architecture relies on the Stardog triplestore, a commercial full-fledged solution that has been adopted in the OntoTrans project. The access to the triplestore is mediated by a module written in Python developed as part of the Tripper interface, created by SINTEF and used within OntoFlowDM (see next section). Tripper is an abstraction layer with the purpose of making the tool flexible and independent from the underlying technology (e.g., Stardog, Fuseki, GraphDB, and others). In this way, other suitable solutions, such as GraphDB or Apache Fuseki, can be easily integrated with minimal effort by developing the relevant backend.

This architecture represents a deviation from the content of the last deliverable, D4.1. Previously, a proxy component developed within the OntoTrans project was employed as a means to access the OntoFlowKB, in a vision to realize a triplestore-as-a-service. In fact, the proxy exposes a set of REST APIs to enable requests from users, interacting with the knowledge base. However, in OpenModel there is no such actor as the OntoTrans users; the knowledge base needs to be accessed to find the best suitable workflows. For this reason, Tripper, which does not expose REST APIs, has been chosen as a better way to interact with the knowledge base, replacing the proxy component.

[1] The highest possible reasoning level depends on the specific triplestore technology which is being used as backend. Not every triplestore may have the same reasoning levels available, e.g. OW-DL may be not implemented.


© OpenModel Consortium CONFIDENTIAL

## 3.1 TRIPPER AND STARDOG BACKEND

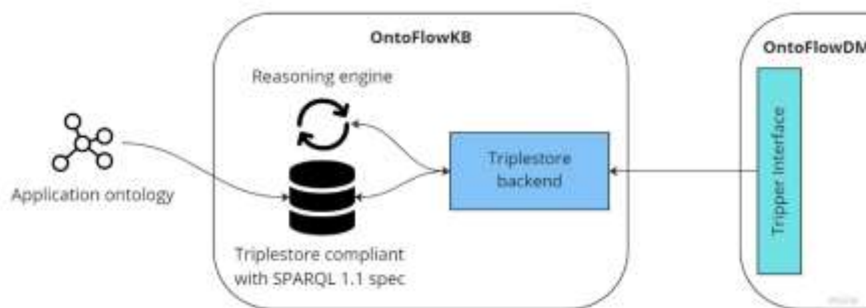



**Figure 3 - Tripper integration in OntoFlow architecture**

Different triplestores usually expose the same set of primary features but the way these are invoked may be slightly different across vendors. In this scenario, the aim of Tripper (Figure 3) is to provide a comprehensive and general set of APIs, which may be considered a standardization effort, so that leveraging an instance of such an interface makes the access to triplestore transparent to the actual vendor-specific implementation.

From a technical point of view, Tripper defines an interface, inspired from rdflib, that encompasses the main operations that it is possible to perform on a triplestore. The interface is based on several abstractions that include the concepts of Namespace, Triple and Database (intended as an independent workspace isolated from the others). In this context, the main component on which the tool relies is the Triplestore class, which provides an initial and basic reference implementation of the interface and delegates to specific modules, called backend, the task of implementing the specific logic of the chosen triplestore. This structure makes Tripper a modular tool, meaning that, by implementing the defined interface, developers can support new triplestores or extend the functionality of existing ones.

| Method signature | Description |
|---|---|
| triples( triple ) | Returns all the matching triples |
| add_triples( triples ) | Add a sequence of triples |
| remove( triple ) | Remove all matching triples |
| create_database( database_name ) | Create a new database in the triplestore |
| remove_database( database_name ) | Remove a database in the triplestore |
| list_database() | List of all databases |
| query( query_string ) | Submit a SPARQL-compliant query |
| update( update_string ) | Submit a SPARQL-compliant update query |
| bind( prefix, iri ) | Bind a new namespace in the database |


© OpenModel Consortium CONFIDENTIAL

| | |
|---|---|
| namespaces() | List of all namespaces in the database |
| serialize( destination, format ) | Get a serialization of the database content at the specified destination |
| parse( source, location, data, format ) | Add new data in the database by parsing triples as File-like objects, URLs and strings |

**Table 1 - Interface defined by the tripper**

As described in the previous section, the MVP architecture of OntoFlow makes use of Stardog as a triplestore and it was, therefore, necessary to implement a tripper-compliant backend to allow the interaction of OntoFlowDM with the knowledge base present in OntoFlowKB. Since the used Trippier version (v0.2.0) does not allow the use of backends that are defined outside the tool, an independent fork was created to implement the interface for Stardog (Stardog Strategy).

The Stardog Strategy makes use of two main libraries to implement the tripper interface: SPARQLWrapper[2] and PyStardog[3]. The former is a Python module that wraps a SPARQL-compliant endpoints to uniformly structure and remotely execute queries; it helps in creating query invocation and in formatting result in a more manageable format and it has been used for all those methods that involve a submission (e.g. triples, add_triples, etc.). Since Stardog has two different endpoints for read-only queries and update ones, the backend handles two different SPARQLWrapper instances and each method is in charge of swapping to the correct ones depending on its implementation and semantics. On the other hand, PyStardog is a library that acts as a client for the Stardog Knowledge Graph and helps with the management of both ontology, databases, and administrative tasks. It has been used for all the rest of the methods that were not involved with the submission of query or required the knowledge of Stardog-specific API (e.g. bind, serialize, parse, etc.).

# 4 ONTOFLOW ARCHITECTURE: ONTOFLOWDM

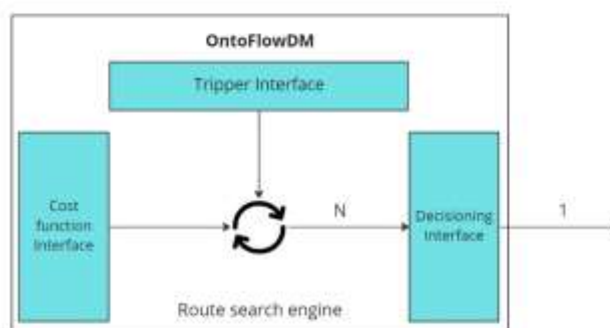



**Figure 4 - OntoFlowDM component overview**

[2] https://sparqlwrapper.readthedocs.io/
[3] https://pystardog.readthedocs.io/


© OpenModel Consortium CONFIDENTIAL

OntoFlowDM, depicted in Figure 4, is the decision-making engine for the OntoFlow architecture. Its main task consists in suggesting the best ontological route that can obtain the desired target and, at the same time, optimize the interested criteria. Different paths may reach the same result but differ in several other aspects (e.g. computational effort, result accuracy, memory occupation, etc.). To achieve this result, the engine performs several steps:

1. Retrieval of all the semantic data contained in the triplestore. This step involves all the information related to the EMMO and the application-specific ontology. The result will represent the search space to explore for the mapping routes.
2. The engine runs the search algorithm to navigate the ontologies and find all the possible routes producing the targeted output. In this process, the engine is also in charge of tagging each step with a cost indication.
3. The mapping routes represent the input for the final phase in which the MCO, through the implementation of tree traversal methods or the employment of machine learning techniques, can find the best route according to predefined criteria (decisioning phase). This step may iteratively use several different optimizers to skim all the route up to get the final one (user-defined choices can also be involved).

At the end, the engine result is an ontological description of the workflow that describe the process to move from the input concepts to the target one. It is noteworthy to point out that such a description is a general-purpose representation of the process. In fact, the benefits of such a high-level description relates not only the output itself. One of the most important additional benefits is that it can be used to automatically generate the documentation of the process. By providing a clear and data-agnostic representation of the flow, it can help to explain how the data is being transformed and what step are involved to obtain the final instance.

From a design perspective, OntoFlowDM is based on the definition of three different interfaces that abstract the three phases described above. The usage of interfaces provides extreme flexibility due to the possibility of dynamically changing the engine's behavior at runtime and relying on a description of the logic rather than on specific implementations. The interfaces present in the module for the current MVP regard the triplestore access, the cost functions definition, and the decisioning logic. Next, a brief explanation of the characteristics of these interfaces and how they have been implemented from a practical perspective.

As detailed in the previous section, the triplestore interface is implemented employing Tripper, which provides a tool able to handle the triplestore-specific logic and access and query the knowledge base contained in OntoFlowKB. The MVP's implementation currently uses a Stardog backend but the approach is general and other triplestores can be easily integrated by creating the proper tripper-compliant backends.

**Figure 5 - Example of YAML descriptor file for the definition of cost functions**


© OpenModel Consortium CONFIDENTIAL

The current cost functions interface, provides a way to define the logic with which the engine tags the cost attribute to each step. The configuration is fed into the engine by means of a YAML file that handles all the specific details of the logic and is loaded at the initialization phase. In the MVP architecture, the cost is evaluated based on the properties and relations of nodes in a graph. The configuration file may list all relevant triple predicates and assign a fixed cost to each one. Additionally, in some cases, the cost may be based on the specific value of a predicate. Figure 5 shows an example of the YAML file: the left image is representative of the first case, so a cost of 5.0 is assigned each time a node has the property identified by the namespace; in the second image, the predicate2 can have different costs based on the actual value of the relation.

It is important to note that certain relationships are mandatory for traversing the graph and moving from one concept to another, such as the **mapsTo**, **instanceOf**, and **subClassOf** predicates. These relationships are essential for the navigability of the graph and must always be present in the configuration in order to accurately evaluate their cost.

The third interface concerns the optimization algorithms to search for the best final route to select. This step consider the KPIs, defined at the beginning of the process, that describe the metrics to optimize (e.g. computational effort, memory occupancy, result accuracy, etc.). The way to do this, is provided to the engine by means of the Strategy pattern that allows to create custom modules to implement the custom logic for specific uses cases. Each module must implement the same interface that will be used after the discovery of all the possible routes to select only the feasible one.

Figure 6 shows the interface class that must be used as a reference for the implementation of custom strategies. The main method, **get_best_routes**, takes all the routes obtained during the previous steps and implement a tree-traversal logic to choose the best one. More effort will be expended in the coming months to define a more general interface that also takes into account the MCO modules developed in the other tasks.

```
class MCOStrategy():

  def __init__(self):
    pass

  def get_best_route(self, routes):
    pass
```

**Figure 6 - MCO Interface definition**

From a practical point of view, the engine can be instantiated as shown in Figure 7, the constructor takes in three parameters, that is, a binding for each interface described. From a practical point of view, the process of route searching can be initiated by calling the "method" method, which takes as input the IRI of the desired output and input elements (more details will be given in the example section).

```
engine = OntoFlowDMEngine(triplestore = <triplestore_instance>, cost_file = <cost_definition_file>, mco_interface = <mco_strategy>)
routes = engine.getmappingroute(
    meta=<output_iri>,
    instances=[<input_instances>],
    ...
)
```

**Figure 7 - Usage of the OntoFlow engine**



© OpenModel Consortium CONFIDENTIAL

The current implementation of the engine makes use of the Tripper also for the route-searching algorithm that is realized inside the module. Furthermore, the algorithm exposed by the Tripper has two customizable features: the first one regards the possibility to overwrite the cost that is used for the navigation predicate, in the specific case of OntoFlow this has been exploited by substituting the cost values with the one defined inside the cost descriptor file. On the other hand, the algorithm allows customizing the structure of the nodes to be used for the composition of the route. **OntoFlowMappingStep** and **OntoFlowValue** are the two structures created for OntoFlow-specific purposes (see YAML file generation in the example).

# 5 YAML GENERATION

The output of OntoFlow is the ontological description of a workflow that describes, as a set of concepts and relationships between them, the final process by which the output can be derived from the available ontology. This description, being ontologically defined, contains no information either about the actual data to be used or about how to perform the various functions. For these reasons, the output as it is, cannot be executed directly by ExecFlow, since it is still in abstract form.

To solve this problem, it is necessary to convert the ontology description of the workflow into a declarative description. This type of description highlights what computational elements are involved in the process by defining what input nodes, computational tasks, and how the data flow goes from one node to another. Again, since OntoFlow does not possess the information about the concrete data to be used, such a description will mainly contain references and access metadata to retrieve the necessary information. Concretely, such a description is realized with a configuration file in YAML format, generated by OntoFlow from the final route chosen after the optimization process. The YAML file is the starting point for ExecFlow to compose and execute all the blocks necessary to create the final result.

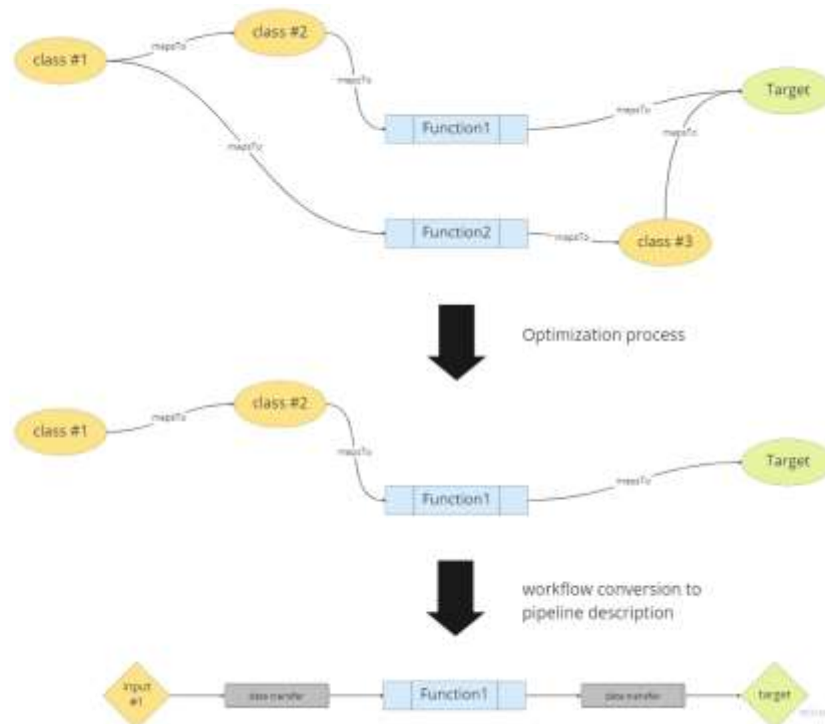



**Figure 8 - Example of the steps taken by OntoFlow. The first two steps represent route search and selection of the best one. Below, the best route is converted into a pipeline description that describes the input elements and computational nodes.**


© OpenModel Consortium CONFIDENTIAL

# 6 ONTOFLOW IN ACTION: DATAMODEL GENERATION

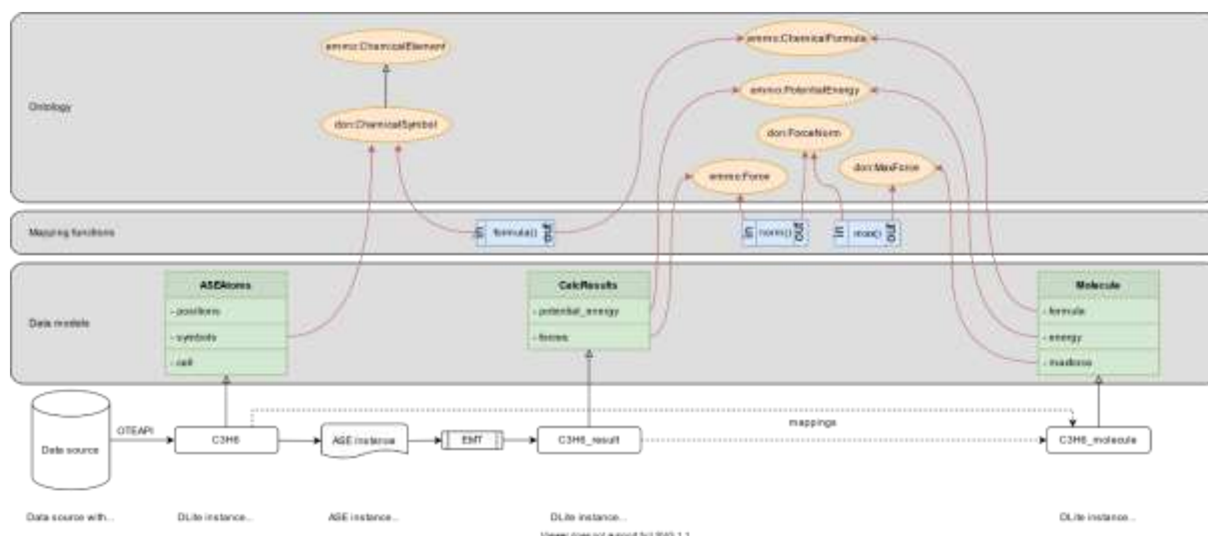



**Figure 9 – Scenario high-level overview**

In this section, we showcase a concrete application of OntoFlow with respect to the different elements that has been presented in this document. The use-case regards the automatic generation of instances of datamodels starting from data already present in the platform.

In particular, the scenario involves two different datamodels: ASEAtoms and CalcResult. ASEAtoms contains information about the atoms in a molecule, their element, position and symbols, while CalcResult contains information about the calculated properties of a molecule, such as its energy and maximum force. The scenario also assume the existence of two different instances of these models that constitutes the input data for the example. On the other hand, the Molecule datamodel, which has the fields formula, energy and maxforce, represents the final output and the schema of the final instance that we want to generate. Intuitively, this can be done by navigating the relations in the ontology backwards to find the combinations of operations that allow to compute each of the fields of the final datamodel.

For the sake of clarity, w focus only on the workflow regarding the formula property and, in particular, we simulate the presence of two different ways of reaching it, respectively, the first one with higher accuracy of the result but also higher computational effort, and the second one representing the inverse case.

## 6.1 PREPARATION PHASE: POPULATE THE KNOWLEDGE BASE

The initial step to demonstrate the process is to fill OntoFlowKB with all the semantic information previously discussed. Specifically, it is necessary to have the information related to the application ontology, its mappings to the datamodels properties and the computational element (ontological functions). In particular, the IRI of the application ontology for this case is [http://example.com/demo-ontology#](http://example.com/demo-ontology#).


© OpenModel Consortium CONFIDENTIAL

This step can be performed using the tripper interface that, by acting as an intermediary with the triplestore, is able to store all the information described above.

```
ts = Triplestore(backend="stardog", base_iri="http://localhost:5820", database="ontoflow")
DON = ts.bind("don", "http://example.com/demo-ontology#")
```

**Figure 10 - Creation of the tripper interface for stardog backend and creation of the namespace for the example ontology**

It also provides a method to create an ontological mapping of Python functions by defining a node for the function itself, its input argument and its output. To simulate the two different paths, we will insert two different functions (Figure 11) for the Formula field characterized by different levels of accuracy and computational effort.

```
func1_iri = ts.add_function(
    get_formula,
    expects=[DON.Symbols],
    returns=[DON.Formula],
)

ts.add([DON.Formula, DON.hasAccuracy, DON.Low])
ts.add([DON.Formula, DON.hasComputationalEffort, DON.Low])
```

```
func2_iri = ts.add_function(
    get_formula_complex,
    expects=[DON.SymbolsComplex],
    returns=[DON.Formula_Complex],
)
ts.add([DON.Formula_Complex, DON.hasAccuracy, DON.High])
ts.add([DON.Formula_Complex, DON.hasComputationalEffort, DON.High])
```

**Figure 11 - The add_function method is utilized to declare that the function 'get_formula' has an input that corresponds to the Symbols concept and an output associated with the Formula concept. Same for the 'get_formula_complex' function. The difference of input/output in the declaration of the two functions is due to a limitation in the current implementation of OntoFlow.**

The last step consists of adding all the mapping to the properties of both the Molecule and ASEAtoms datamodels.

```
ts.add_mapsTo(DON.Symbols, ASEAtoms, "symbols")
ts.add_mapsTo(DON.Formula, Molecule, "formula")

ts.add_mapsTo(DON.Formula_Complex, Molecule, "formula")
ts.add_mapsTo(DON.SymbolsComplex, ASEAtoms, "symbols")
```

**Figure 12 - Mapping of the datamodels properties to the demo ontology. The current implementation requires the properties to be linked to the output values**

To have a clear vision of the situation, Figure 13 depicts the final ontological graph resulting from these operations.

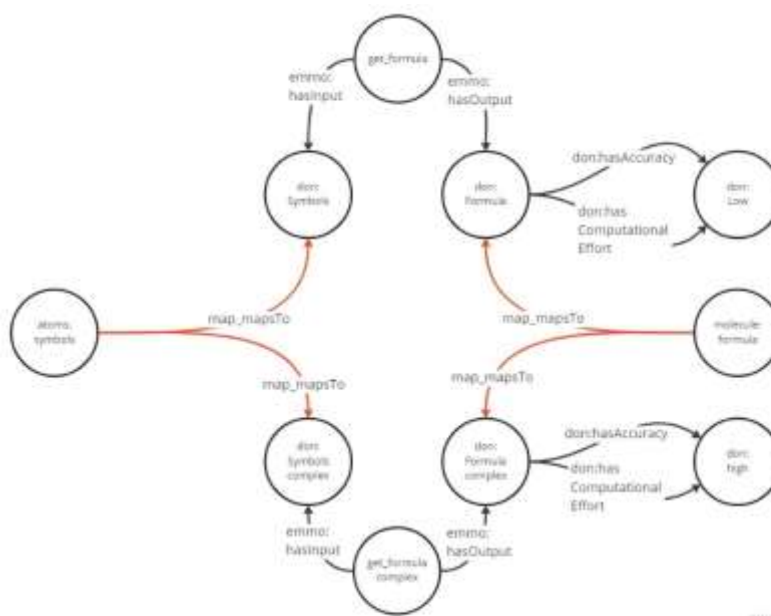



**Figure 13 - Search space for the mapping routes**

## 6.2 CONFIGURATION PHASE: ONTOFLOWDM ENGINE CONFIGURATION

Once all the ontological elements have been set up in the triplestore, the next step is to configure OntoFlowDM's engine to navigate the ontology graph and find all the best routes to get the molecule formula. As described in the previous sections, the engine is defined in terms of three interfaces, so its configuration mainly consists of injecting the proper components. It is straightforward that the triplestore instance defined in Figure 8 perfectly fits for populating the first interface of the engine.

Figure 15 shows the YAML file used for defining the second interface of the engine, related to the computation of costs. It is possible to note that the configuration contains all the mandatory predicates for the navigability of the graph (**mapsTo**, **subClassOf**, **instanceOf**) and other predicates of interest for the specific application. In this case, a hasMemoryOccupancy predicate has been used to declare the memory occupancy of input data (symbols) and a proper cost has been defined for each of its possible values.

The last step involves the definition of the MCO logic for the choice of the final route. For the sake of simplicity, in this example, we decided to use an MCO logic that returns the first mapping explored.

```
engine = OntoFlowDMEngine(triplestore = ts, cost_file = r"C:\Users\cost_definitions.yaml", mco_interface = first_route)
```

**Figure 14 - Configuration of the OntoFlowDMEngine**


© OpenModel Consortium CONFIDENTIAL

```yaml
function_costs:
  predicate:
    - mapsTo:
      namespace: "http://emmo.info/domain-mappings#mapsTo"
      cost: 3.0
      self-contained: true
    - function:
      namespace: "https://w3id.org/function/ontology#Function"
      cost: 10.0
      self-contained: true
    - instanceOf:
      namespace: "http://emmo.info/datamodel#instanceOf"
      cost: 1.0
      self-contained: true
    - subClassOf:
      namespace: "http://www.w3.org/2000/01/rdf-schema#subclassOf"
      cost: 1.0
      self-contained: true
    - label:
      namespace: "http://www.w3.org/2000/01/rdf-schema#label"
      cost: 1.0
      self-contained: true
```

```yaml
- hasAccuracy:
  namespace: "http://example.com/demo-ontology#hasAccuracy"
  self-contained: false
  values:
    - value: "http://example.com/demo-ontology#High"
      cost: 5.0
    - value: "http://example.com/demo-ontology#Low"
      cost: 10.0
- hasComputationalEffort:
  namespace: "http://example.com/demo-ontology#hasComputationalEffort"
  self-contained: false
  values:
    - value: "http://example.com/demo-ontology#High"
      cost: 20.0
    - value: "http://example.com/demo-ontology#Low"
      cost: 10.0
```

**Figure 15 - YAML file for the definition of costs. It contains both mandatory predicates for the exploration of the graph (left) and custom ones for the specific use case(right).**

## 6.3 EXECUTION PHASE: SELECTION OF THE BEST ROUTES

The next step in the process is to run the OntoFlowDM engine to search for all mapping routes within the triple-store. This process, triggered by the "getmappingroute" method in Figure 14, takes as input the IRI of the datamodel to be instantiated (in this case Molecule) and the available input instances. The latter parameter is useful for figuring out which input IRIs (atmos:symbols) will then be the final nodes of the mapping routes.

```python
molecule = engine.getmappingroute(
    meta=Molecule,
    instances=[inst, result],
    quantity=ureg.Quantity,
)
```

**Figure 14 - Execution of the engine**

Figure 15 shows the routes that were identified by the process: as expected, the engine identified two routes, each of which uses a different function to arrive at the final result. In the graph you can also see the costs associated with each step calculated from the cost descriptor given as input.


© OpenModel Consortium CONFIDENTIAL

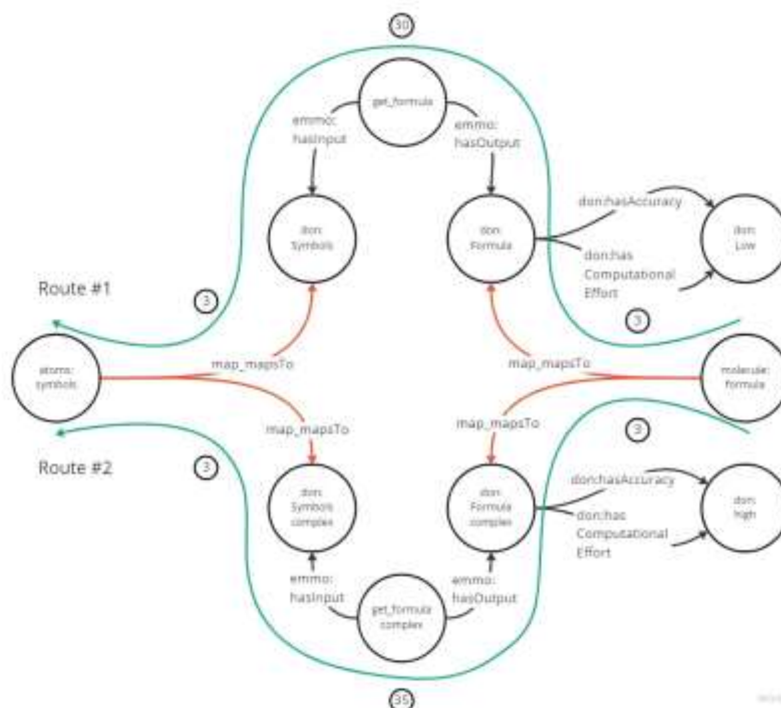



Figure 15 - Mapping route found by the engine

The last step of the engine is to perform an optimization and skimming operation of the mapping routes to identify the best one. As described earlier, this process is based on the definition of an MCO strategy that, in the specific case of the 'example, implements a path-finding algorithm of the lowest cost. From Figure 15, it is evident that the least cost route that will be chosen is number 1.

## 6.4 FINAL PHASE: GENERATION OF YAML FILE

The last phase of the process consists of creating the proper YAML file to input the ExecFlow component. This operation is handled by OntoFlow which exploits the tree structure of the returned route to delegate the creation of the different parts of the file to the nodes themselves. Each node (class **OntoFlowMappingStep**) defines a recursive method **_to_yaml** that is in charge of creating the file portion related to its subtree and it is recalled on each of its children nodes.

In the case of the example, the file generated by OntoFlow is the one depicted in Figure 16. It is possible to note that the 5-steps ontological route has been compacted into three main steps. The first one, the **execflow.pipeline** for symbols, declares the pipeline responsible for retrieving, reading, and mapping the input data. For the MVP implementation, we assume that the pipeline is already defined in some databases and referenced inside the ontology. The second step relates to the execution of the function '**get_formula**' which has as input the data of the previous step (a DLite collection with label '**symbols'**) and whose output will be stored in the execution context with the '**get_formula_output**' name. The final step has the same semantics as the first one and it aims to read data from the context and map them to the final structure of the output.


© OpenModel Consortium CONFIDENTIAL

```yaml
steps:
- workflow: execflow.pipeline
  inputs:
    pipeline: file://random/path/pipeline_for_symbols.yaml
    run_pipleline: get_symbols
- calcjob: openmodel.get_formula
  inputs:
    input1: '{ get_dlite_istance_by_label("symbols") }'
  postprocess:
  - '{ ctx.current.outputs["output_data"] | to_ctx("get_formula_output") }'
- workflow: execflow.pipeline
  inputs:
    pipeline: file://random/path/pipeline_for_formula_output.yaml
    run_pipleline: get_formula_output
```

**Figure 16 - YAML file with pipelines descriptions**

# 7 CODE REFERENCES

This section provides all the reference links to the repositories for the OntoFlow components and the related example:

- OntoFlowKB: https://github.com/H2020-OpenModel/OntoFlowKB
- OntoFlowDM: https://github.com/EMMC-ASBL/OntoFlow
- Tripper with Stardog backend: https://github.com/EMMC-ASBL/tripper/tree/deliverable_4.2



© OpenModel Consortium CONFIDENTIAL

## ACKNOWLEDGMENT

*This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 953167.*

*This document and all information contained herein is the sole property of the OpenModel Consortium. It may contain information subject to intellectual property rights. No intellectual property rights are granted by the delivery of this document or the disclosure of its content.*

*Reproduction or circulation of this document to any third party is prohibited without the consent of the author(s).*

*The content of this deliverable does not reflect the official opinion of the European Union. Responsibility for the information and views expressed herein lies entirely with the author(s).*

*All rights reserved.*