Preprint Cambridge Centre for Computational Chemical Engineering ISSN 1473 – 4273

An agent composition framework for the J-Park Simulator - a knowledge graph for the process industry

Xiaochi Zhou[1], Andreas Eibeck[1],

Mei Qi Lim[1], Nenad B. Krdzavac[1], Markus Kraft[1,2,3]

released: 13 May 2019

[1] CARES
Cambridge Centre for Advanced Research and
Education in Singapore,
1 Create Way,
CREATE Tower, #05-05,
Singapore, 138602

[2] University of Cambridge,
Department of Chemical Engineering
and Biotechnology,
Philippa Fawcett Drive ,
Cambridge, CB3 0AS
United Kingdom
E-mail: mk306@cam.ac.uk

[3] Nanyang Technological University,
School of Chemical and
Biomedical Engineering,
62 Nanyang Drive,
Singapore,
637459

Preprint No. 227

UNIVERSITY OF CAMBRIDGE

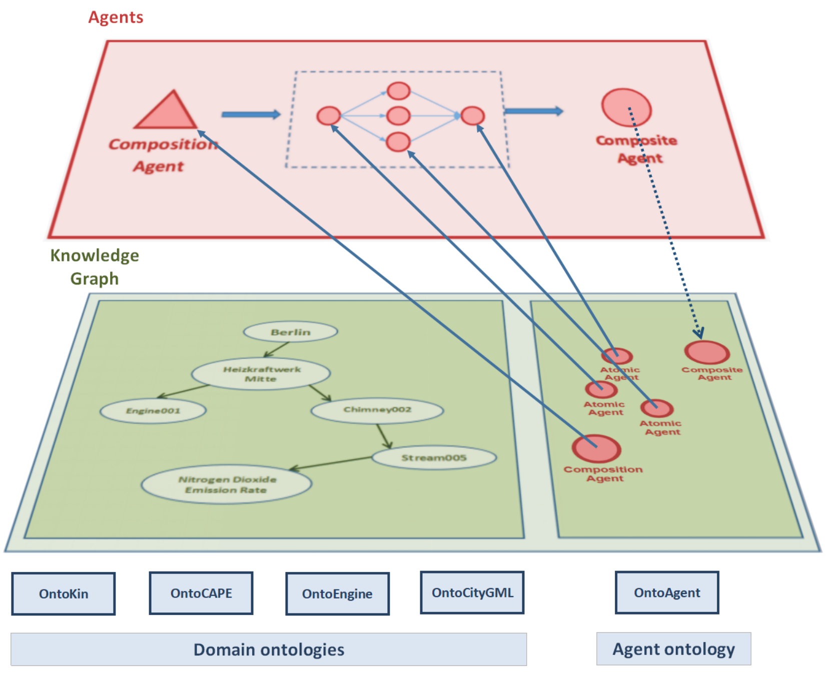
Keywords: Process industry, Knowledge graph, Semantic Web, Semantic Web Service Composition, Agent, Cross-domain, Linked Data

Edited by
Computational Modelling Group
Department of Chemical Engineering and Biotechnology
University of Cambridge
West Cambridge Site
Philippa Fawcett Drive
Cambridge CB3 0AS
United Kingdom

Fax:	+ 44 (0)1223 334796
E-Mail:	c4e@cam.ac.uk
World Wide Web:	`http://como.cheng.cam.ac.uk/`

Abstract

Digital twins, Industry 4.0 and Industrial Internet of Things are becoming ever more important in the process industry. The Semantic Web, linked data, knowledge graphs and web services/agents are key technologies for implementing the above concepts. In this paper, we present a comprehensive semantic agent composition framework. It enables automatic agent discovery and composition to generate cross-domain applications. This framework is based on a light-weight agent ontology, OntoAgent, which is an adaptation of the Minimal Service Model (MSM) ontology. The MSM ontology was extended with grounding components to support the execution of an agent while keeping the compatibility with other existing web service description standards and extensibility. We illustrate how the comprehensive agent composition framework can be integrated into the J-Park Simulator (JPS) knowledge graph, for the automatic creation of a composite agent that simulates the dispersion of the emissions of a power plant within a selected spatial area.



Highlights

- The light-weight ontology, OntoAgent, has been developed based on MSM ontology.
- An agent composition framework based on OntoAgent has been developed.
- A cross-domain air pollution scenario is used to illustrate the agent composition framework.

Contents

1 Introduction

An eco-industrial park (EIP) [27] aims at an industrial symbiosis that promises improvement of energy and resource efficiency as well as reduction of environmental impact. Numerous studies have been carried out focusing on resource networks within a single domain such as water [17, 18, 36], energy [1, 22, 39], and material [3, 10, 34]. However, in an EIP, symbiotic relationships do not only exist within single domain network as resource networks and entities across domains are intertwined and affect each other. In order to achieve Pareto optimality among different domains, all domains need to be taken into consideration simultaneously. Consequently, tools to simulate, analyze, optimize, and coordinate heterogeneous components across multiple domains (e.g. to simulate a chemical plant's material production and consumption, and analyse its effect on the energy network) are necessary. The establishment of such tools clearly requires the integration of data and software tools from relevant domains. However, the integration is challenging due to the friction of communication between different domains. For example, the term "vessel" in the chemical engineering domain usually means pressurized container and yet refers to a large boat in the transportation domain. Besides the communication friction, due to the heterogeneity of data formats and conventions across domains, there is also the lack of a uniform access to data.

The concept of a cross-domain knowledge graph has been identified as one of the solutions to alleviate the communication friction and to provide uniform data access. The knowledge graph is essentially an interconnected collection of terminologies and statements across domains [4]. It stores and connects data semantically, i.e. each distinct concept, instance, and relation is described by an unique Uniform Resource Identifier (URI)[1] (e.g. ontocape:Vessel[2] for pressurized container and dbr:Vessel_(boat)[3] for boat). The unique mappings between URIs and concepts or instances lead to explicitness and disambiguation of information. A collection of explicit declaration of concepts is referred to as an ontology [35], and the set of tools and methods to process and utilize such semantic data is regarded as semantic technology. The disambiguation makes the information in the knowledge graph formal, i.e machine-readable. Therefore, the semantic knowledge graph could avoid the friction of cross-domain communication with the unambiguity of information. Meanwhile, the formality of data enables uniform access to them through queries constructed in query languages such as SPARQL[4]. We have already implemented the J-Park Simulator (JPS), a cross-domain knowledge graph for the process industry, which includes ontologies in domains such as chemical process engineering, chemical kinetics, internal combustion engines, etc.[4].

The dynamic nature of an eco-industrial park, requires the knowledge graph describing such entities to cope with this aspect. Consequently, the knowledge graph must use components that reflect and/or effect changes in the graph over time, e.g. constantly update data and maintain the knowledge graph structure. In this paper, we refer to these compo-

[1]https://www.w3.org/Addressing/URL/uri-spec.html

[2]http://www.theworldavatar.com/ontology/ontocape/chemical_process_system/CPS_realization /plant_equipment/apparatus.owl#Vessel

[3]http://dbpedia.org/resource/Vessel_(boat)

[4]https://www.w3.org/TR/rdf-sparql-query/

nents as agents. We also defined the term "agent" in this paper to refer to applications and web services that utilize semantic technologies and are accessible on World Wide Web. Currently, there is a number of agents updating the JPS knowledge graph. For a cross-domain knowledge graph where the contributors typically come from diverse professional backgrounds, it is a good strategy to lower the barrier for creating new agents in order to encourage its adoption and curb its investment cost.

Furthermore, in cross-domain scenarios, there will be simulation or optimization tasks that require the consecutive execution of multiple agents, for example, the output of an agent that simulates engine emissions is used as the input of an agent that models the dispersion profile of the emission stream. In order to fulfill complex objectives such as control and optimization, agents must be able to communicate and hence coordinate with each other. At the moment, the coordination between the agents for the JPS knowledge graph is hard-coded by developers. The hard-coded coordination is time-consuming to implement and lacks flexibility in a dynamic environment. Semantic technologies have been long applied for automatic coordination between agents [31], and such coordination is also known as semantic-based agent[5] composition. Semantic-based agent composition could automatically interpret the functions and interfaces of agents, and plan their coordination for achieving complex goals, on top of the machine-readable agent descriptions. Moreover, a complete automated agent[5] composition process also includes an execution phase, in order to put the coordination plan in use [31].

The semantic description of agents[5] is necessary for semantic-based agent[5] discovery, composition, and automated execution. Meanwhile, with the semantic descriptions, the agents could be also represented in the knowledge graph so that the knowledge graph has an uniform management for both data and agents. To model the descriptions, an agent[5] ontology is necessary. Two most prevailing agent[5] ontologies are Web Service Modeling Ontology (WSMO) [6] and OWL-S [19], which are well-established and expressive, coming with software tools for agent[5] discovery, composition, and execution. WSMO describes an agent's capability, non-functional properties, interface and goal. OWL-S [19] which is built on the Web Ontology Language (OWL), contains components including profile, processes, and groundings. In the context of agent ontologies, grounding is the linking between semantic and syntactic information. Typically, the serialization of an HTTP request follows a certain syntactic format; therefore, a mapping is needed to convert the semantic data into such a syntactic format. Such a mapping is an example of the grounding. Although these two models could comprehensively describe agents[5] and their goals, they are not favored by the knowledge graph due to their heavy weight. Clearly, an increased model complexity increases the cost for developers to adopt.

The semantic community have created some lightweight solutions. For example, the agent description of Semantic Annotations for WSDL and XML Schema (SAWSDL) [15] is minimal i.e. it does not directly define how agents are described, and only annotates components in a Web Services Description Language (WSDL) description. WSDL is an XML-based interface description language to describe agents on a syntactic level[6]. In other words, SAWSDL depends on WSDL for execution hence the communication is stan-

[5]"agent" here refers to "web service"; however, in this paper, the two terms are interchangeable. For the consistency, "web service" is replaced by "agent"

[6]https://www.w3.org/TR/2001/NOTE-wsdl-20010315

dard specific. WSMO-lite [14] is another minimized agent ontology to annotate WSDL descriptions. Compared to SAWSDL, WSMO-lite provides richer information outside the WSDL but its grounding is still restricted to WSDL. Another lightweight agent ontology is hRESTs [16], which describes RESTful agents[5], i.e. agents[5] that follow the Representational State Transfer (REST) architecture style [7]. Minimal Service Model (MSM) is an agent ontology that is not specific for any communication standards. MSM [28] only captures the common components of the mainstream models above-mentioned; this ontology could be extended with other ontologies for additional description, e.g. including the information for invocation. The purpose of this design is to maintain the compatibility with existing standards such as WSDL, WSMO, and OWL-S. However, MSM's grounding mechanisms do not fit the agents which have adopted the lightweight communication standard. Therefore, a lightweight agent ontology suitable for describing agents in the above-mentioned knowledge graph is currently absent.

An agent[5] composition framework is required for implementing the agent composition and discovery. However, most of the existing agent composition frameworks are designed for heavy agent ontologies such as WSMO and OWL-S. For example, SOA4All [20] proposed a framework for working with DAML-S, which was later superseded by OWL-S. Sirin et al. [33] developed a framework with the hierarchical task network (HTN) planner SHOP2 [24]. It works with agents described by OWL-S. The composition framework OWLS-XPlan [12] also works with OWL-S. Fujii and Suda [8] introduced a framework that uses Component service Model with Semantics (CosMoS), as an agent[5] model, which is also not considered lightweight. To the best of our knowledge, Rodriguez-Mier et al. [30] developed the only known composition framework based on a lightweight ontology (MSM). However, this framework does not include the execution function, which is vital for completing the composition process. Therefore, a complete agent composition framework with the execution function and compatible with a lightweight agent ontology is currently absent as well.

The **purpose of this paper** is to introduce and describe the implementation of a comprehensive agent composition framework that leverages semantic technologies for automatic agent discovery and composition to generate cross-domain application. The paper includes the following:

- The introduction and description of OntoAgent, an ontology for describing agents, which is an extension of MSM. With its light weight, OntoAgent lowers the cost of creating agent instances in the cross-domain knowledge graph.
- The introduction and description of the agent composition framework which is based on OntoAgent, and consists of agent composition, discovery, and execution functionalities. Such a framework enables the knowledge graph to coordinate agents and execute them automatically. To the best of our knowledge, this is the first agent composition framework working with a lightweight agent ontology that supports execution functionality.
- The illustration of the unique agent composition framework in the context of the JPS along with a cross-domain air pollution scenario.

The remaining parts are structured as follows. Section 2 gives an overview of the JPS, which is the research platform for implementing the agent composition framework. Section 3 describes the development of OntoAgent. Section 4 presents the implementation of the unique agent composition framework. Section 5 illustrates how the agent composition framework can operate in the JPS for the automatic creation of a cross-domain composite agent that simulates the dispersion profile for a power plant within a selected area. Section 6 discusses the limitation of the current work and provides suggestions for improvement. Section 7 outlines the conclusions for this paper.

2 J-Park Simulator

The JPS is a platform where components across domains share a common ground for data management and semantic interoperability between each other.

Ontologies play pivotal roles in the JPS project. Ontologies from different domains offer formal definition of concepts and relations in a certain field, the JPS project has been developing and integrating the ontologies systematically. For example, OntoCAPE [21] is a large-scale ontology for chemical process engineering, and it is the starting ontology for JPS. OntoCAPE is then extended into OntoEIP [40], describing the eco-industrial parks and their networks. Meanwhile, OntoCityGML, which is a semantic upgrade of CityGML [9], is integrated to describe 3D models and other properties of buildings and landscapes. OntoKin [5] is an ontology developed for chemical kinetics and provides specification for chemical species and mechanisms. OntoEngine[7] specializes in describing the operation of internal combustion engines, it specifies fuel used by the engine as well as the corresponding combustion chemistry model.

The JPS builds a cross-domain knowledge graph following the linked data principle, so that it could be deployed in a distributed fashion across the Web. Each host in this distributed structure stores a part of the knowledge graph and works as an independent authority to control its own data. Moreover, agents update the structure and data of the knowledge graph to reflect the dynamic nature of systems such as eco-industrial park or smart grid.

Before the outcome of this paper is implemented in the JPS, the agents in the JPS were simply software tools represented as agents[5]. To lower the barrier for creating agents, the JPS agents use a lightweight communication standard that constructs HTTP requests with JSON objects in key-value pairs. Due to the absence of semantic description, the agents were not part of the knowledge graph and the coordination between agents was hard-coded by developers. Figure 1 illustrates the components of the JPS so far.

This paper extends the JPS by integrating agent ontology to describe agents as well as implementing the composition framework to automate the coordination between them.

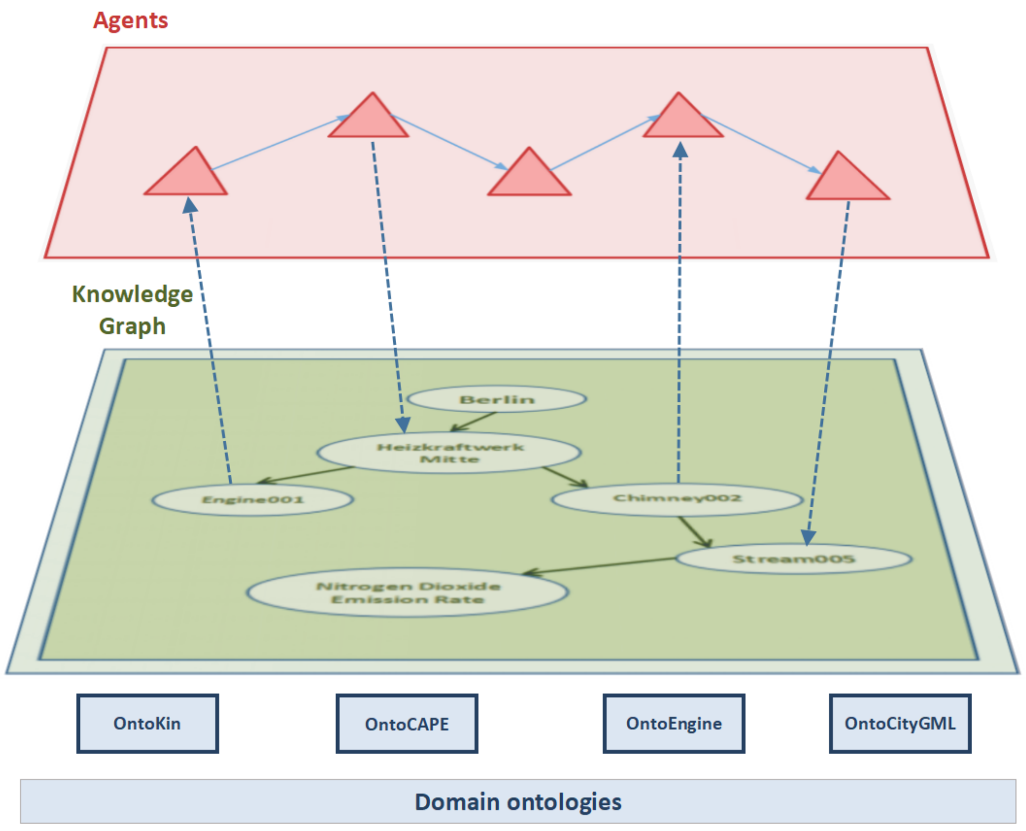


Figure 1: *The JPS knowledge graph and agents: the original status of the JPS is that the knowledge graph (green layer) contains the terminologies (blue boxes) and instances (light green nodes) of domain ontologies. On the agent layer (red layer), the agents (red triangles) read data from the knowledge graph and update it (dotted arrows). The agents cooperate with each other as well (solid arrows).*

3 OntoAgent

To better fit the specific requirements for the agent ontology in the context of a knowledge graph, we customized the MSM ontology into OntoAgent. The role of OntoAgent is to provide machine-readable descriptions of agents for automated operation of agents, including agent discovery, composition, and execution on top of an underlying cross-domain knowledge graph.

OntoAgent utilizes the skeleton of the MSM ontology and adds OWL classes and properties for grounding to support the invocation of the agents as part of an agent composition framework. The extensions and their purposes are described in Table 1 while Figure 2 illustrates the structure of OntoAgent. Appendix A.4 provides detailed information on the property restrictions of OntoAgent.

[7] http://www.theworldavatar.com/ontology/ontoengine/OntoEngine.owl

Table 1: *Extension made upon the MSM agent ontology and their descriptions*

OWL Class	Description
ontoagent:Invocation	To be the container of the invocation information. OntoAgent may integrate more information for invocation, this class provides clear separation of such information.
OWL Properties	**Description**
ontoagent:hasInvocation	To connect the invocation information to the operations.
ontoagent:hasHttpUrl	To define the HTTP address for invoking a certain service of an agent.
ontoagent:hasKey	To define the name of the key in the key-value pair that contains the input JSON Object in the HTTP requests
ontoagent:isArray	To declare whether the I/O parameter is an array of class defined by ontoagent:hasType.
ontoagent:hasType	To directly connect the I/O parameters with ontology classes.

The intention of adding grounding elements to MSM is not to create yet another grounding standard but to capture the most common and fundamental elements of grounding shared by the mainstream standards. Such a design will enable the OntoAgent to support the execution of agents in the JPS cross-domain knowledge graph while keeping the extensibility and flexibility of MSM.

One key question for a minimal agent ontology is whether it provides sufficient and necessary information to support each phase of the agent composition process, including agent discovery, composition, and execution. OntoAgent has inherited the IO (Input and Output) model from MSM instead of the IOPE (Input, Output, Precondition, and Effect) model used by ontologies such as OWL-S [19], i.e. OntoAgent mainly describes the input and output parameters of agents to represent them. Arguably, due to the reduction of precondition and effects, the model loses its expressiveness for describing activities such as booking a ticket, where the criteria of invoking an agent largely depend on the precondition of this agent, which is affected by the effects of the preceding agents. However, in our context, the majority of tasks focus on simulations and optimization, which are data-centric, in the sense that whether an agent meets its invocation requirement depends on the inputs it receives. In a data-centric scenario, the input/output (I/O) parameters of an agent are sufficient to define the function of an agent hence support function-based discovery and composition in most of the cases. The outcome from Rodriguez-Mier et al. [30] also supports this argument.

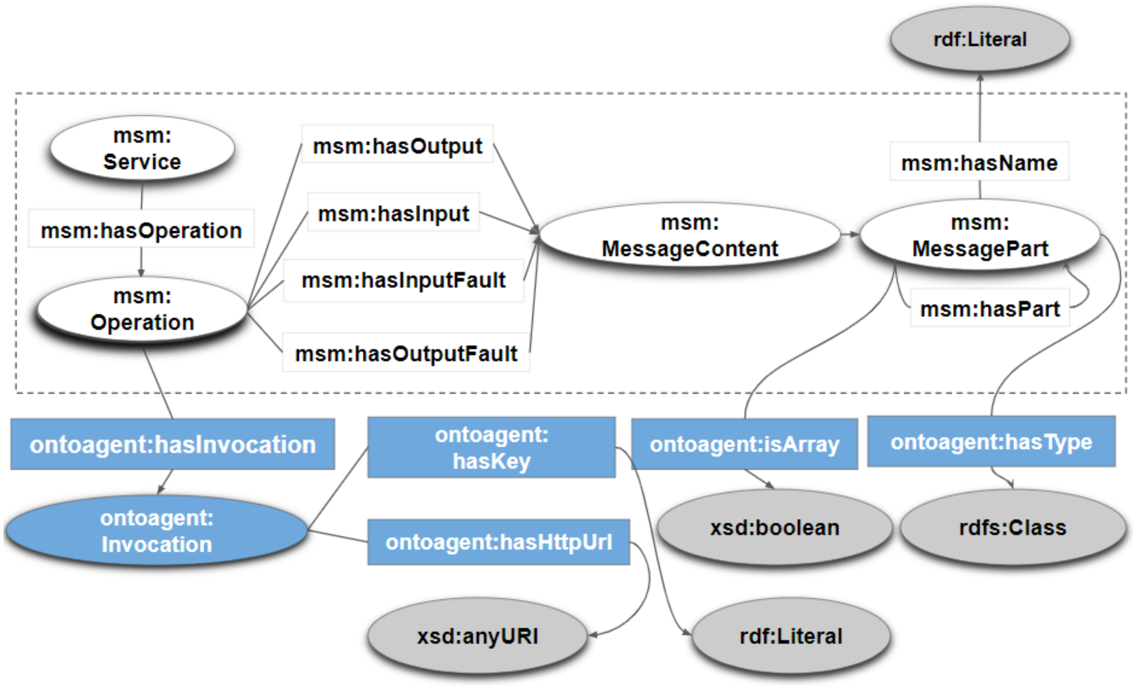


Figure 2: *The design of OntoAgent: the ovals denote the classes and the arrows with annotation denote properties, of OntoAgent. The components within the dotted box are the native MSM classes and relations, while those outside the box, with the name-space ontoagent are newly defined in OntoAgent.*

However, it is evident that the description of I/O data types could not differentiate agents such as division agent and multiplication agent which have the same data flow but different purposes. Nevertheless, in a cross-domain environment, where the tasks for agents are very specific (e.g. to calculate the emission of an internal combustion engine), agents with identical data-flow are rare. In future, concept specifications of finer granularity (i.e. finer subdivision of concepts) could alleviate the problem.

For the execution of an agent, the basic grounding information provides the most essential information for invocation: where to send the HTTP request and how to structure the input. Such a grounding enables the implementation of an execution agent that is standard neutral but potentially compatible with mainstream standards, in the context of the cross-domain knowledge graph. The detail of the invocation mechanism will be discussed in Section 4.2.

4 The agent composition framework

The purpose of implementing an agent composition framework is to fulfill tasks that require the consecutive execution of more than one agent, without hard-coded coordination. An agent composition framework creates plans for agent coordination in an automated and dynamic fashion, hence increase the efficiency and flexibility of coordinating agents.

The composition framework we designed contains two agents: the composition agent and the execution agent. The composition agent takes the user requirement and creates the composite agents. The other component of the composition framework, the execution agent takes the description of composite agent and concrete input values as inputs and executes the agents constituting the composite in sequence. Figure 3 demonstrates the complete process of agent composition including the execution of the composite agent and this section will introduce the implementation details of the composition agent and the execution agent respectively.

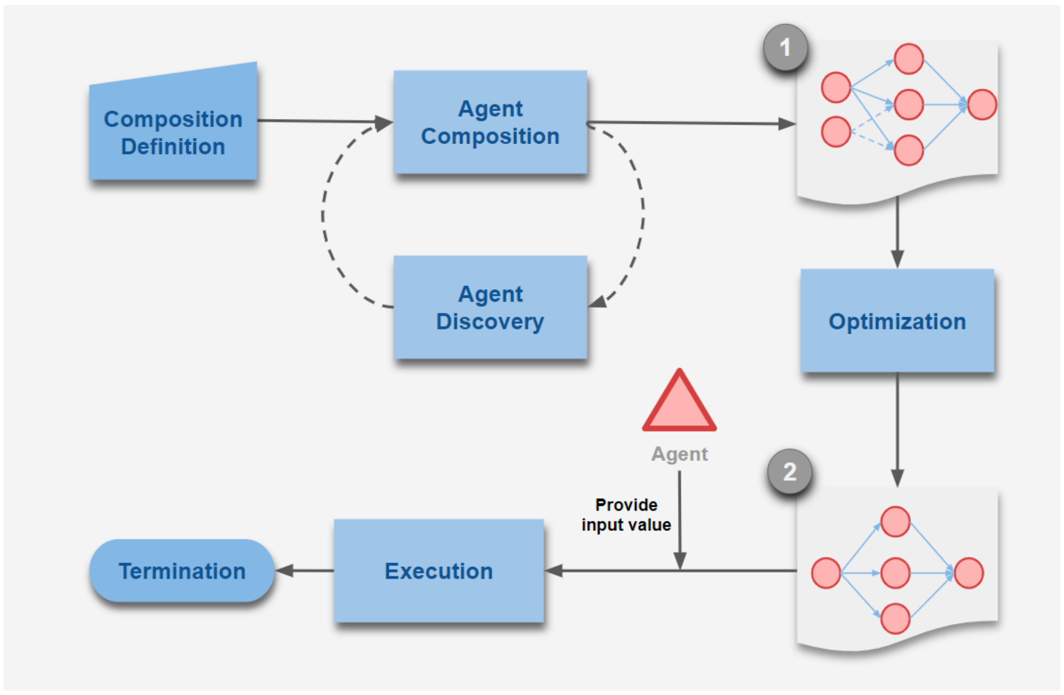


Figure 3: *The process of the agent composition implemented: Each blue panel denotes a phase in the composition process. Solid arrows represent the process sequence and the dotted ones are iterative sequence. The panels containing agents (red nodes) represent composition results: ① is the composition result with multiple solutions; ② is the optimized composite agent. The composition process starts from defining the requirements for the composite agent, and ends with the execution of the composite agent. The execution will be triggered when an agent provides the input values.*

4.1 The composition agent

The composition process starts from defining the requirements for the composite agent by specifying the types of the I/O parameters, in the form of URIs[8]. The definition could come from either a human user or an agent (for demonstration purpose in the use case, some extra components are implemented to support human users). The discovery module

[8]https://www.w3.org/Addressing/URL/uri-spec.html

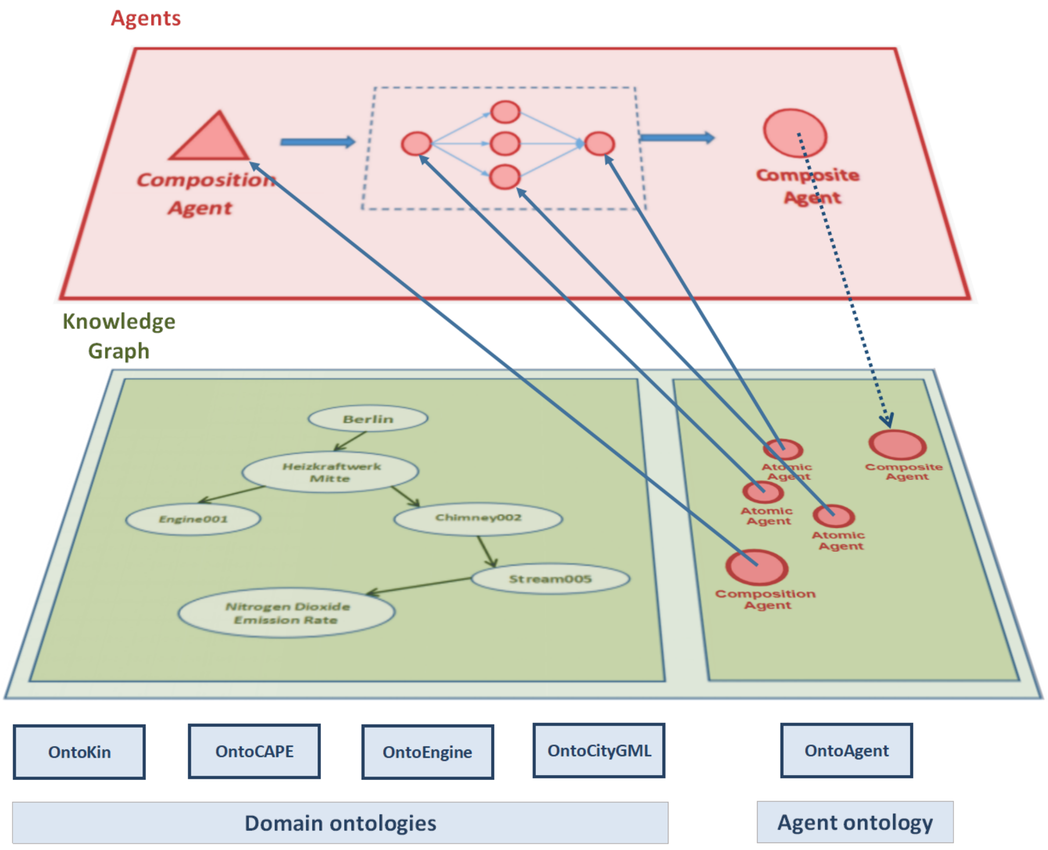


Figure 4: *Knowledge graph integrated with OntoAgent and the composition agent: now knowledge graph is populated with the OntoAgent ontology and its instances(red nodes). Agents in action are represented by red triangles. The agents layer (red layer) demonstrates the composition agent creating composite agents out of atomic agents. The dotted arrow denotes the composition agent adding the new composite agent to the knowledge graph. The solid arrows denote the connection between agent instances in the knowledge graph and the agents in action on the agent layer.*

within the composition agent locates agents within the knowledge graph, that meet the I/O requirements via a SPARQL query and reasoning (reasoning is not yet implemented in the proof-of-concept prototype). The composition module works with the discovery module iteratively to come up with the composition plan. In order to better work with agents described by OntoAgent, the composition agent adopts a common graph-based composition approach which utilizes the matching of semantic input-output parameters to arrange sequences of agents. Such an approach has been widely applied for agent[5] composition [2, 11, 13, 23, 25, 26, 29, 32, 37]. The essence of graph-based composition is to append agents which fulfill the input requirements provided either by initial inputs or outputs of other agents already appended to the composition result. The graph-based composition algorithm repeats the process of appending new eligible agents until all the initially required outputs for the composite agent are achieved. When all the required outputs are achieved or the process takes longer than the preset time-out value, the process of composition terminates. Figure 4 illustrates how the composition agent creates a composite agent on top of the knowledge graph and algorithm 1 in the Appendix A.1 introduces the composition algorithm in detail. In this algorithm, function ***discover_agent*** discovers all the agents that are eligible for the composition. In other words, it returns agents of which all inputs could be fulfilled by the inputs collected so for. Appendix A.2 shows the simplified Java implementation of the function ***discover_agent*** while Appendix A.5 illustrates the implementation with a flowchart.

The iterative phases of agent discovery and composition yield one or more plans for the agent coordination. Due to the existence of alternative solutions, the framework will need to select out the optimal one. Therefore the process proceeds to the optimization phase. The optimization module essentially eliminates the redundant agents when multiple ones are providing the same data. In this implementation, the optimization is based on Quality-of-Service (QoS), which reflects the performance of an agent. For now, the scores are set by the developer. After the optimization, the optimal composition result will be created. The result will be serialized in JSON format and stored. After that, whenever an execution is triggered by either a human user or an agent, the composition process proceeds to the execution phase.

4.2 The execution agent

The execution agent is a part of the agent composition framework. It takes composition result as input, executes each atomic agent and feeds their outputs to the downstream agents, according to the execution sequence stored in the composition result. It could execute a single atomic agents as well.

The execution agent supports the invocation of agents described by OntoAgent but remains potentially compatible with other standards. This is one of the major distinction of our agent composition framework. As shown in Figure 5, the execution phase closely works with the knowledge graph. In this phase, the execution agent reads the semantic descriptions of agents within the serialized composition result, from the knowledge graph. Appendix A.3 shows the simplified Java source code for the execution agent and Appendix A.6 demonstrates the execution process with a flowchart. During the execution of an atomic agent, the agent takes data from the knowledge graph and updates the knowledge graph with the new data produced.

The execution agent is customized to work with grounding information provided by OntoAgent, Figure 6 explains how the execution agent utilizes the grounding information for invocation. Firstly, with DataType properties ontoagent:isArray, msm:hasName, and ontoagent:hasType alongside with the intrinsic mapping between the name and type, the execution agent converts the output of the upstream agent into a JSON object that the downstream agent accepts. Secondly, based on the properties ontoagent:hasHttpUrl and ontoagent:hasKey, the execution agent constructs the HTTP request with a key-value pair.

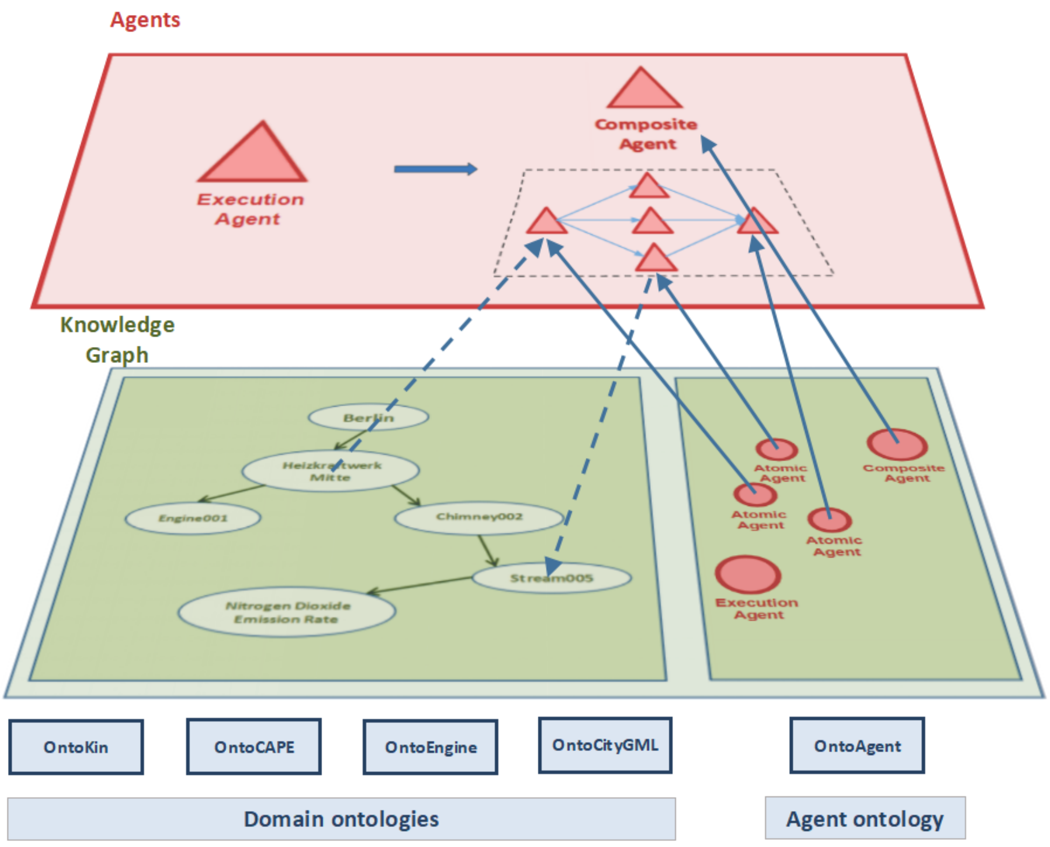


Figure 5: *The execution of a composite agent: the solid arrows mark the connection between the descriptions of the agents(red nodes) in the knowledge graph and the implementation of agents in action (red triangles). The upward dotted arrows denote the reading from the knowledge graph while the downward one depicts the writing to the knowledge graph.*

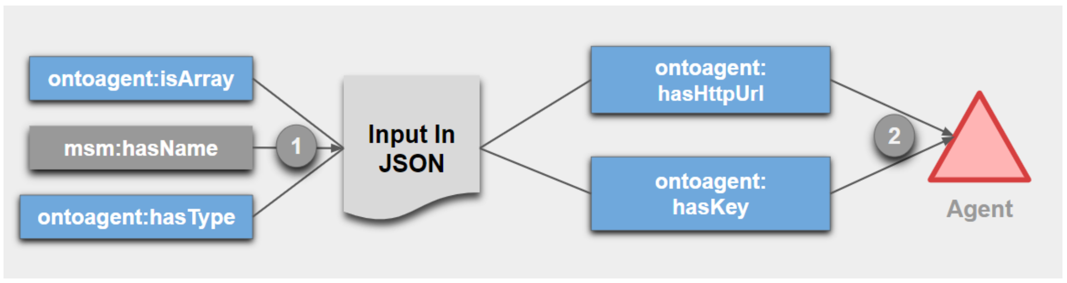


Figure 6: *The execution agent's invocation of an agent with OntoAgent description: step ① utilizes **ontoagent:isArray**, **msm:hasName**, and **ontoagent:hasType** to construct a JSON object containing all the input data for invoking the agent. Step ② builds the full HTTP Request containing the input JSON object, based on **ontoagent:hasHttpUrl** and **ontoagent:hasKey**, and sends the concrete request to the agent.*

5 Use case

The OntoAgent ontology and the comprehensive agent composition framework are integrated into the JPS. In this section, we illustrate how the agent composition framework automates the creation of a cross-domain composite agent that simulates the dispersion profile of the emission from a power plant within a selected area. This scenario considers multiple domains such as urban landscape, meteorology, and chemical kinetic reaction mechanisms. It serves as an example of an integrated analytical application that is based on the integration of data and software tools from various domains. This composition agent could potentially be used to assist in evaluating the suitability of proposed location for a new power plant installation, with regards to the potential air pollution impact it could have on the proximity.

5.1 Agents in the JPS knowledge graph

The JPS knowledge graph has been populated with a number of agents described by OntoAgent and eight of them are relevant in this use case:

- City query agent: This agent returns the URI[9] in DBpedia ontology in a selected region. In the background, the agent requests Google Geocoding API[10] and gets the city name e.g. "Berlin", then through DBpedia Ontology Lookup service[11], it retrieves the URI based on the city name.
- Plant query agent: This agent has the same input as the city query agent. It queries the JPS knowledge base and returns the URIs of all the power plants, described by the "PowerPlant" class[12] from OntoCAPE.
- Weather agent: There are three different weather agents for real-time weather data of a selected city in order to demonstrate the optimization phase. The three weather agents use Accuweather, YahooWeather, and OpenWeatherMap respectively. The output weather condition is described by the WeatherOntology[13].
- Building query agent: This agent takes both city and region as input and returns URIs of buildings instances of OntoCityGML ontology by querying the JPS knowledge graph.
- SRM agent: This agent wraps up SRM Engine Suite, a commercial software for the simulation of exhaust emission from internal combustion engines (ICE), as an agent. It takes the URI of reaction mechanism instance of OntoKin and the URI of engine instance under OntoEngine as inputs and produces instances of OntoCAPE "NonReusableWasteProduce" class.

[9] e.g. http://dbpedia.org/resource/Berlin for Berlin

[10] https://developers.google.com/maps/documentation/geocoding/start

[11] https://wiki.dbpedia.org/lookup

[12] http://www.theworldavatar.com/ontology/ontocape/chemical_process_system /CPS_realization/plant.owl#Plant

[13] https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/WeatherOntology.owl

- ADMS Agent: Atmospheric Dispersion Modelling System (ADMS)[14] is another commercial software integrated into the JPS platform as an agent. This agent simulates the dispersion of the pollutant given the weather condition, the dimensions of surrounding buildings, and the details of the emission stream. Currently, there is an absence of specific ontological vocabulary to describe the dispersion; therefore, we use class "Table[15]" to annotate the dispersion grid that is in the tabular form.

5.2 Demonstration

This subsection demonstrates how the above-mentioned composite agent is created through the agent composition framework implemented in the JPS. A series of screen-shots will illustrate the steps of the composition process from defining of the composite agent to its execution[16].

As shown in Figure 7, the framework provides a graphical user interface (GUI) for users to define a composite agent following the OntoAgent model, which includes components such as operation, message content and message parts. The user could add components to the composite agent using the plus buttons on each component. When a user presses a plus buttons on a message part box (highlighted by the red rectangle), an Ontology Lookup Interface (OLI) shown in Figure 8 will pop up for the user to define the ontology class connected to this message part.

Due to the difficulty for human users to type URIs, an OLI is implemented to search for URIs of ontology classes. The OLI loads a mapping between the natural language label of an ontology class and its URI[17] into an Apache Solr[18] supported text search engine, so that searching the term "plant" or "power plant" will return a series of URIs including the URI for the ontology class power plant. For this use case, the inputs are defined as "OntoKin:ReactionMechanism" and "OntoCityGML:EnvelopeType" and the output to be "csvw:Table".

After defining the ontology classes of each message part, the user could press the compose button in Figure 7 to start the composition process, which is supported by the algorithm demonstrated in Appendix A.1. When the composition framework comes up with the composition result, it shows the visualization of the composition result illustrated by Figure 9. When the user presses the "Select Optimal Path" button, the framework will optimize this composition result by eliminating agents with a lower score. The framework then presents the optimal composition result as shown in Figure 10. By pressing the "Send to executor" button, the user could proceed to the execution of the composite agent. The implementation of agent execution is demonstrated in Appendix A.3.

For the execution phase, the framework provides an integrated GUI for data input and output visualization. Figure 11 demonstrates the execution of the use case composite

[14]http://www.cerc.co.uk/environmental-software/ADMS-model.html

[15]https://www.w3.org/ns/csvw#Table

[16]The agent composition framework is accessible via http://www.theworldavatar.com/JPS_COMPOSITION/

[17]e.g. The class "http://www.theworldavatar.com/OntoEIP/OntoEN/power_plant.owl#PowerPlant" has a property "rdfs:label", of which value is the text string "power plant"

[18]https://lucene.apache.org/solr/

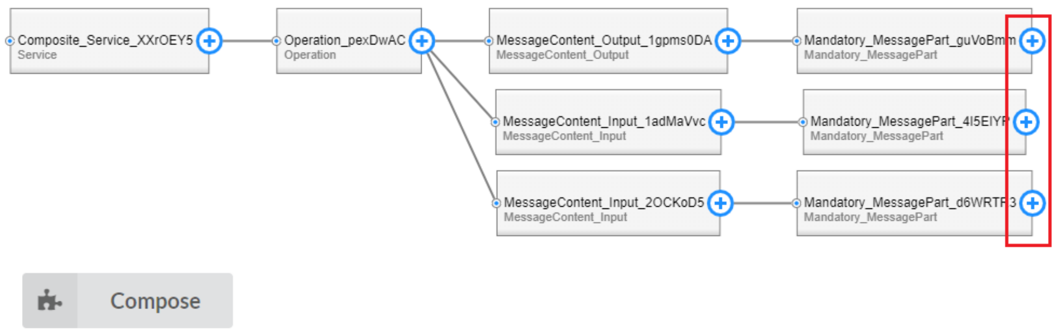
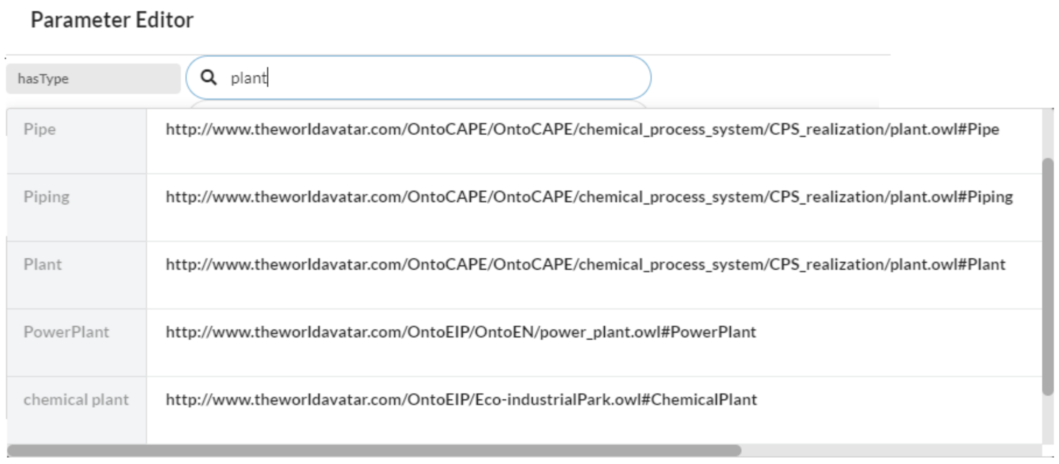

Composite_Service_XXrOEY5
Service
Operation_pexDwAC
Operation
MessageContent_Output_1gpms0DA
MessageContent_Output
MessageContent_Input_1adMaVvc
MessageContent_Input
MessageContent_Input_2OCKoD5
MessageContent_Input
Mandatory_MessagePart_guVoBmm
Mandatory_MessagePart
Mandatory_MessagePart_4I5EIYR
Mandatory_MessagePart
Mandatory_MessagePart_d6WRTR3
Mandatory_MessagePart
Compose

Figure 7: *GUI for defining a composite agent: the hierarchical structure reflects the OntoAgent agent model and the boxes denote the components of OntoAgent such as service, operation, message part, and message content. The plus buttons on each component allows users to add more next-level components and hence to adjust the number of inputs and outputs. Meanwhile, by selecting any box and pressing delete, the user could also delete a component. In this use case, the composite agent defined has two inputs and one output. For simplicity, hasInputFault and hasOutputFault properties are removed. If click on the plus button on the message part box, an Ontology Lookup Interface (OLI) will pop up and allow the user to define the ontology classes of inputs and outputs. After defining the classes of all inputs and outputs, the user could use the compose button to trigger the composition process.*


Parameter Editor

hasType | plant

Pipe	http://www.theworldavatar.com/OntoCAPE/OntoCAPE/chemical_process_system/CPS_realization/plant.owl#Pipe
Piping	http://www.theworldavatar.com/OntoCAPE/OntoCAPE/chemical_process_system/CPS_realization/plant.owl#Piping
Plant	http://www.theworldavatar.com/OntoCAPE/OntoCAPE/chemical_process_system/CPS_realization/plant.owl#Plant
PowerPlant	http://www.theworldavatar.com/OntoEIP/OntoEN/power_plant.owl#PowerPlant
chemical plant	http://www.theworldavatar.com/OntoEIP/Eco-industrialPark.owl#ChemicalPlant

Figure 8: *Ontology lookup service: this GUI allows the users to define the composition requirements by converting natural language terms into ontology classes.*

agent. When the user finishes entering all the inputs, the framework will execute the composite agent and then visualize the execution result in the same GUI.

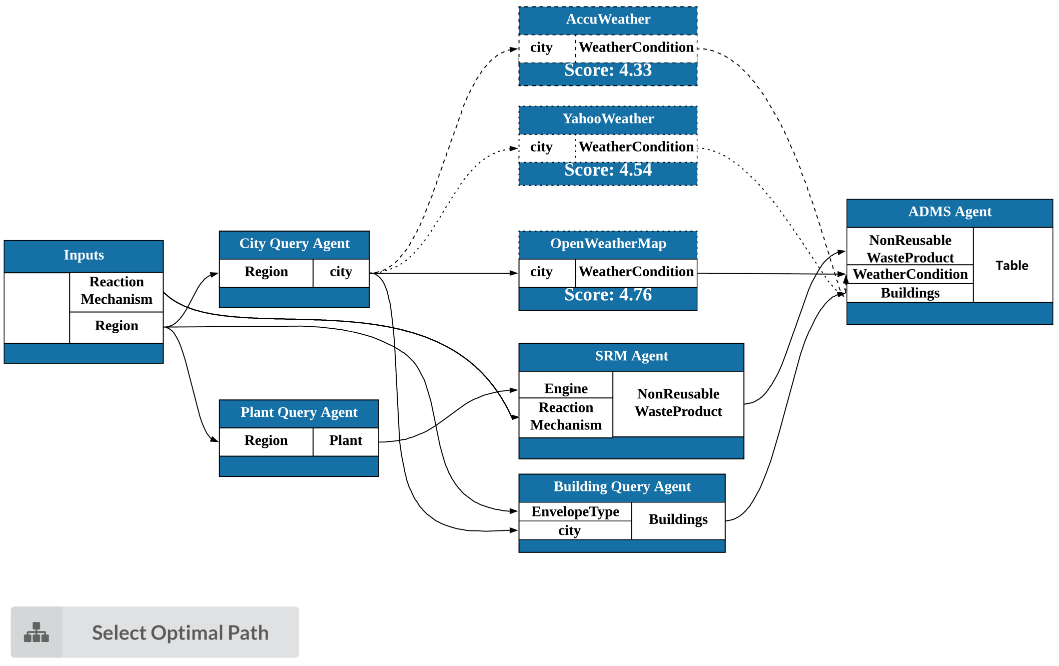


Figure 9: *Visualization of composition result for the use case: this use case requires a composite agent that takes reaction mechanism and region as inputs and produces air dispersion simulation result (temporarily represented by Table class). Each blue and white box denotes an agent, the annotation on its sides are short terms for the I/O types. Arrows represent data flow between the agents. This composition result gives three alternatives for weather data, the weather agents connected with dotted arrows are to be eliminated due to their lower performance scores (scores are currently defined by the developer).*

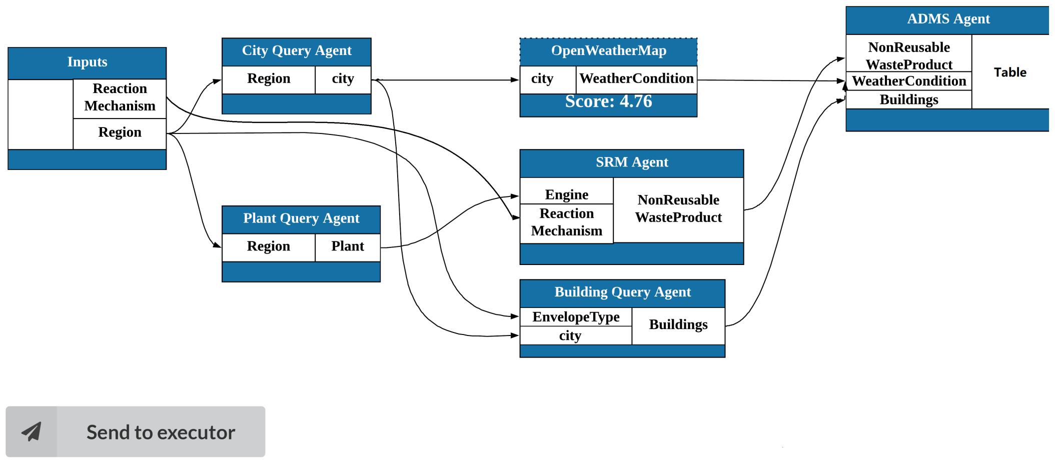


Figure 10: *Visualization of optimized composition result for the use case: the two weather agents with a lower QoS score have been removed from the composition result and hence the composition result is optimized.*

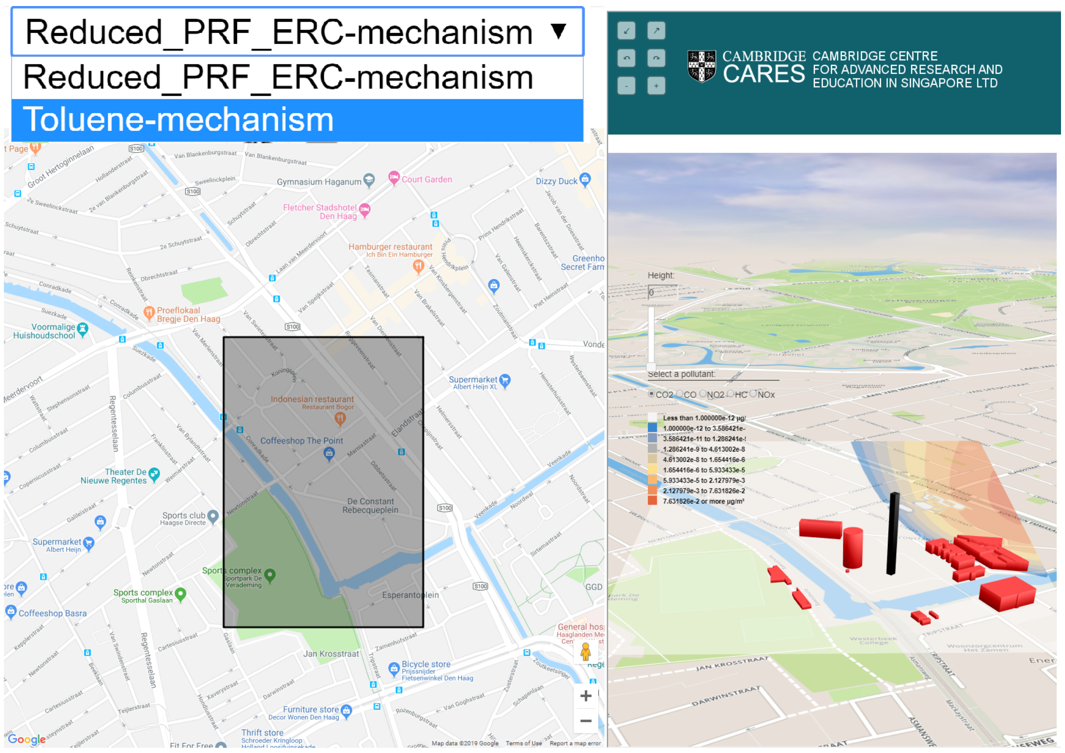

Reduced_PRF_ERC-mechanism ▼
Reduced_PRF_ERC-mechanism
Toluene-mechanism
CAMBRIDGE CARES
CAMBRIDGE CENTRE FOR ADVANCED RESEARCH AND EDUCATION IN SINGAPORE LTD
Height:
0
Select a pollutant:
● CO2 ○ CO ○ NO2 ○ HC ○ NOx
Less than 1.000000e-12 μg/
1.000000e-12 to 3.586421e-
3.586421e-11 to 1.286241e-
1.286241e-9 to 4.613002e-8
4.613002e-8 to 1.654416e-6
1.654416e-6 to 5.933433e-5
5.933433e-5 to 2.127979e-3
2.127979e-3 to 7.631826e-2
7.631826e-2 or more μg/m³

Figure 11: *Visualization of execution result: on the left is the sub-screen for inputs, including the drop-down list for specifying the reaction mechanism and the map for selecting the region. On the right is the visualization for the output, which is the air dispersion grid.*

6 Limitations and Outlook

The present implementation of OntoAgent and the agent composition framework have some shortcomings. Firstly, as mentioned, OntoAgent only captures the inputs and outputs of an agent. Such a design limits the range of application for OntoAgent as it does not describe activities such as booking a ticket. However, such a limitation is acceptable for the current status of the knowledge graph, where the number of agents is limited and the function of agents focuses on tasks such as optimization and simulation. In the long run, when more tasks for the agent description emerge, one could easily extend the functionality of OntoAgent with its extensibility. We trust such extensions will not bloat OntoAgent, as the extensions could be designed in a modular way. Those who have the needs to extend OntoAgent only need to learn the module of interest. For example, OntoAgent is not able to describe a composite agent. Consequently, the composite agents created are not yet written into the knowledge graph. In future, we will extend OntoAgent to describe composite agents in a modularized fashion.

Secondly, this paper only introduces the proof-of-concept implementation of the agent composition framework prototype. The evaluation of performance on phases such as discovery, composition, and execution is left aside. However, the purpose of this paper is to present a proof-of-concept design where agent composition framework is integrated with a knowledge graph, increasing the robustness and scalability of this system will be a major focus in the future.

Lastly, the current QoS-based optimization is built upon arbitrary agent performance scores. We are now experimenting with the application of emerging technologies such as blockchain-based smart contracts for agent performance evaluation and record management.

7 Conclusion

This paper presents the lightweight agent ontology OntoAgent that keeps the extensibility and flexibility of MSM but supports grounding for execution which captures the fundamental elements for agent invocation. Its lightweight clearly decreases the cost of creating an agent instance in the knowledge graph. We have also demonstrated that this agent ontology efficiently facilitates the phase of agent composition and execution in the scenario of a cross-domain knowledge graph.

Also, the paper illustrates the implementation of a comprehensive agent composition framework integrated with the execution agent, which works with the lightweight agent ontology OntoAgent. The agent composition framework provides a solution to create and execute composite agents to fulfill complex tasks on top of a cross-domain semantic knowledge graph.

Lastly, the paper demonstrates the integration of OntoAgent and the agent composition framework into the JPS and how the agent ontology and the framework work together upon the JPS knowledge graph and creates a composite agent for the analysis of the air pollution impact from power plants in a selected urban area. In future, we will use the same framework for other implemented use cases, for example waste heat network agent that optimizes a small inter-plant waste heat recovery network to maximize its overall energy efficiency[19] [38], world power plant CO2 calculation agent that estimates the CO2 emission from power plants all over the world using surrogate model[20], and the agent for building management of laboratories[21] that monitors and predicts activities in chemical laboratories.

8 Acknowledgements

This project is supported by the National Research Foundation(NRF), Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise(CREATE) programme. Markus Kraft acknowledges the support of the Alexander von Humboldt foundation.

[19]Accessible via http://www.theworldavatar.com:82/hw

[20]Accessible via http://www.theworldavatar.com/JPS_CO2EMISSIONS/

[21]Accessible via http://www.theworldavatar.com:83/BMSIndoor/

A Appendices

A.1 Graph-based agent composition algorithm

Algorithm 1 Composition Algorithm

```
 1: function Composition($I_0, O_0$)        ▷ I and O denote the user defined I/O parameters
 2:     $G \leftarrow \emptyset$                                  ▷ G: the final composition result
 3:     $C \leftarrow \emptyset$                          ▷ C: the set of all agents discovered
 4:     $D_{collected} \leftarrow I_0$
 5:     repeat
 6:         $i \leftarrow i+1$
 7:         $L_i \leftarrow \emptyset$                              ▷ denotes one layer of agents
 8:         $A \leftarrow \emptyset$          ▷ A: a temporal set for agents discovered in this iteration
 9:         $A \leftarrow$ discover_agent(D_collected)
10:         for all $a = \{I_a, O_a\} \in A$ do
11:             if $a \notin C$ then
12:                 $L_i \leftarrow L_i \cup \{a\}$                    ▷ Push an agent in one layer
13:                 $D_{collected} \leftarrow D_{collected} \cup \{O_a\}$
14:             end if
15:         end for
16:         $C \leftarrow C \cup A$
17:         $G \leftarrow G \cup \{L_i\}$              ▷ The final result G is an ordered array of layers
18:     until ($O_0 \subset D_{collected}$) or time out
19: end function
```

A.2 Agent discovery function implementation in Java

```java
public class AgentDiscovery {

public static ArrayList<String> discover_agent(ArrayList<String> inputs) {
ArrayList<String> agent_iris = new ArrayList<String>();
// Query the SPARQL Endpoint and generate a mapping
// between agents and their input types
Map<String, ArrayList<String>>
agents_and_inputs_mapping = query_sparql_endpoint();

for (Map.Entry<String, ArrayList<String>> entry :
agents_and_inputs_mapping.entrySet()) {
    if (inputs.containsAll(entry.getValue())) {
/* if the agent's inputs is a subset of the inputs required,
   this agent is considered eligible          */
     agent_iris.add(entry.getKey());
     }
}
return agent_iris;
}

public static Map<String, ArrayList<String>> query_sparql_endpoint() {

Map<String, ArrayList<String>> agents_and_inputs_mapping =
new HashMap<String, ArrayList<String>>();

String agent_query_string =
"PREFIX msm:<http://www.theworldavatar.com/ontology/ontoagent/MSM.owl#> " +
"PREFIX ontoagent:<http://www.theworldavatar.com/ontology/OntoAgent.owl#> " +
"SELECT DISTINCT ?agent ?inputType" +
"WHERE " +
" {      " +
"   ?agent msm:hasOperation ?operation ." +
"   ?operation msm:hasInput ?messageCotentsForInput ." +
"   ?messageCotentsForInput msm:hasMandatoryPart ?mandatoryPart ." +
"   ?mandatoryPart ontoagent:hasType ?inputType ." +
"   }";

// The SPARQL query to retrieve the input types of agents
QueryExecution qe = QueryExecutionFactory.sparqlService(
"http://www.theworldavatar.com/damecoolquestion/agents/query",
agent_query_string);
ResultSet results = qe.execSelect();

// Fire the SPARQL query
while (results.hasNext()) {
    QuerySolution result = results.next();
    String agent = result.get("agent").toString();
    String inputType = result.get("inputType").toString();

    if(agents_and_inputs_mapping.containsKey(agent)) {
    agents_and_inputs_mapping.get(agent).add(inputType);
    }
    else {
    agents_and_inputs_mapping.put(agent, new ArrayList<String>());
    }
}

return agents_and_inputs_mapping;
}
}
```

A.3 Agent execution function implementation in Java

```java
public class ExecutionAgent {
/*
 * The method receives the URIs of two consecutive agents and the output
 * for the upstream agent, converts the output of the precedent agent
 * to the format that the subsequent receives as input, and executes
 * the subsequent agent with the formatted input.
 */
public static JSONObject execute_an_agent(String upstream_agent_uri,
        String downstream_agent_uri, JSONObject inputJSON) {
```

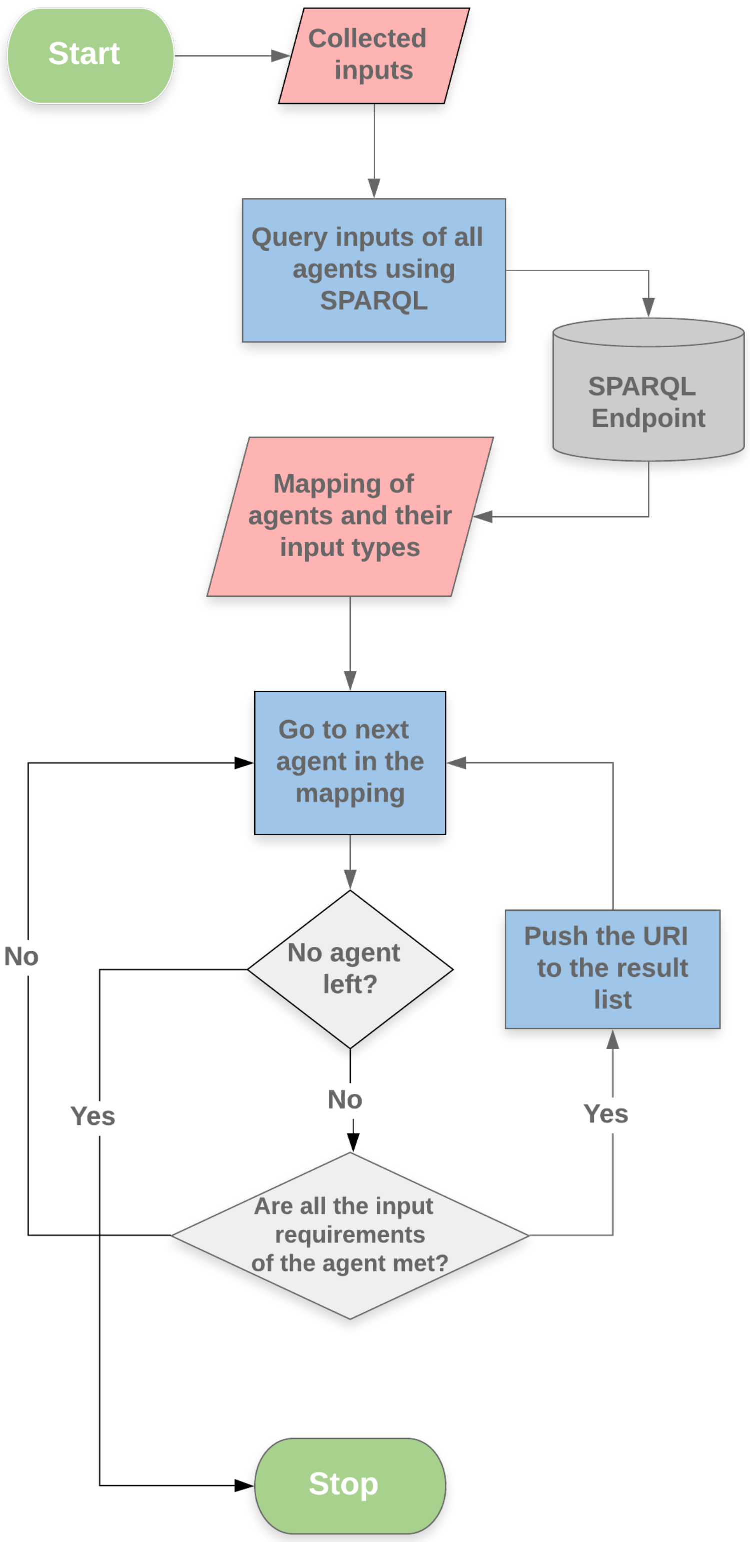
```java

        Map<String, String> name_mapping = generateNameMapping(
                        upstream_agent_uri, downstream_agent_uri);
        JSONObject input_json = mapJSONObject(inputJSON, name_mapping);
        return executeAgent(input_json, downstream_agent_uri);
}

// Generate a mapping between the potentially different keys between the two
// consecutive agents.
public static Map<String, String> generateNameMapping(
        String upstream_agent_uri, String downstream_agent_uri) {

String query_for_downstream_agent_template =
"PREFIX msm:<http://www.theworldavatar.com/ontology/MSM.owl#> "
+ "PREFIX ontoagent:<http://www.theworldavatar.com/ontology/OntoAgent.owl#> "
+ "SELECT  ?type ?key" +
+ "WHERE "
+ "                {     "
+ "          <%s> msm:hasOperation ?operation ."
+ "     ?operation msm:hasInput ?messageCotentsForInput ."
+ "     ?messageCotentsForInput msm:hasMandatoryPart ?mandatoryPart ."
+ "          ?mandatoryPart msm:hasType ?type ."
+ "          ?mandatoryPart msm:hasName ?key ."
+ "  }";

String query_for_upstream_agent_template =
"PREFIX msm:<http://www.theworldavatar.com/ontology/MSM.owl#> "
+ "PREFIX ontoagent:<http://www.theworldavatar.com/ontology/OntoAgent.owl#> "
+ "SELECT  ?type ?key" +
+ "WHERE "
+ "          {     "
+ "          <%s> msm:hasOperation ?operation ."
+ "     ?operation msm:hasOutput ?messageCotentsForOutput ."
+ "     ?messageCotentsForOutput msm:hasMandatoryPart ?mandatoryPart ."
+ "          ?mandatoryPart msm:hasType ?type ."
+ "          ?mandatoryPart msm:hasName ?key ."
+ "  }";

QueryExecution qe_up = QueryExecutionFactory.sparqlService(
                "http://www.theworldavatar.com/damecoolquestion/agents/query",
                String.format(query_for_upstream_agent_template,
                                upstream_agent_uri));
ResultSet results_upstream = qe_up.execSelect();

QueryExecution qe_down = QueryExecutionFactory.sparqlService(
                "http://www.theworldavatar.com/damecoolquestion/agents/query",
                String.format(query_for_downstream_agent_template,
                                downstream_agent_uri));
ResultSet results_downstream = qe_down.execSelect();
return process_query_result_for_mapping(results_upstream,
                results_downstream);
}

public static JSONObject mapJSONObject(
                JSONObject output_from_upstream_agent,
                Map<String, String> name_mapping) {

JSONObject input_for_downstream_agent = new JSONObject();
Iterator<String> keys = output_from_upstream_agent.keys();
while (keys.hasNext()) {
        String key = keys.next();
        String new_key = name_mapping.get(key);
        input_for_downstream_agent.put(new_key,
                        output_from_upstream_agent.get(key));
}

return output_from_upstream_agent;
}

// Construct an HTTP request based on the input JSON Object and the grounding
// information of the agent
public static JSONObject executeAgent(JSONObject input_JSON_object,
                String agent_uri) {

String key = "";
String url = "";
String query =
  "PREFIX msm:<http://www.theworldavatar.com/ontology/MSM.owl#> "
+ "PREFIX ontoagent: <http://www.theworldavatar.com/ontology.owl#>"
+ "SELECT  ?key ?HttpUrl" + "WHERE " + "{      "
+ "  <%s> msm:hasOperation ?operation ."
+ "  ?operation ontoagent:hasInvocation ?invocationContainer ."
+ "  ?invocationContainer ontoagent:hasKey ?key ."
+ "  ?invocationContainer ontoagent:hasKey ?HttpUrl ."
+ "}";

// Make SPARQL query to retrieve grounding information for agent invocation
QueryExecution qe_up = QueryExecutionFactory.sparqlService(
                "http://www.theworldavatar.com/damecoolquestion/agents/query",
                String.format(query, agent_uri));

ResultSet invocation_info = qe_up.execSelect();
while (invocation_info.hasNext()) {
        QuerySolution result = invocation_info.next();
        key = result.get("key").toString();
        url = result.get("HttpUrl").toString();
}
// Construct the HTTP request with information retreived from the semantic
// description of the agent.
URIBuilder builder = new URIBuilder().setScheme("http")
                .setPath(url)
                .setParameter(key, input_JSON_object.toString());

return executeGet(builder);
}

public static Map<String, String> process_query_result_for_mapping(
                ResultSet results_upstream, ResultSet results_downstream) {
```

```java
Map<String, String[]> type_name_mapping = new HashMap<String, String[]>();
Map<String, String> name_mapping = new HashMap<String, String>();
while (results_upstream.hasNext()) {
        QuerySolution result = results_upstream.next();
        String type = result.get("type").toString();
        String name = result.get("key").toString();
        String[] temp = new String[2];
        temp[0] = name;
        type_name_mapping.put(type, temp);
}

while (results_downstream.hasNext()) {
        QuerySolution result = results_downstream.next();
        String type = result.get("type").toString();
        String name = result.get("key").toString();
        type_name_mapping.get(type)[1] = name;
}

for (Map.Entry<String, String[]> entry : type_name_mapping
                .entrySet()) {
        String[] keys = entry.getValue();
        name_mapping.put(keys[0], keys[1]);
}
return name_mapping;
}

// Carry out the HTTP request
public static JSONObject executeGet(URIBuilder builder) {

try {
URI uri = builder.build();
HttpGet request = new HttpGet(uri);
request.setHeader(HttpHeaders.ACCEPT, "application/json");
HttpResponse httpResponse = HttpClientBuilder.create().build()
                .execute(request);
return new JSONObject(
                EntityUtils.toString(httpResponse.getEntity()));
} catch (Exception e) {
}
return null;
}
}
```

A.4 Domain and range restrictions on new roles of OntoAgent

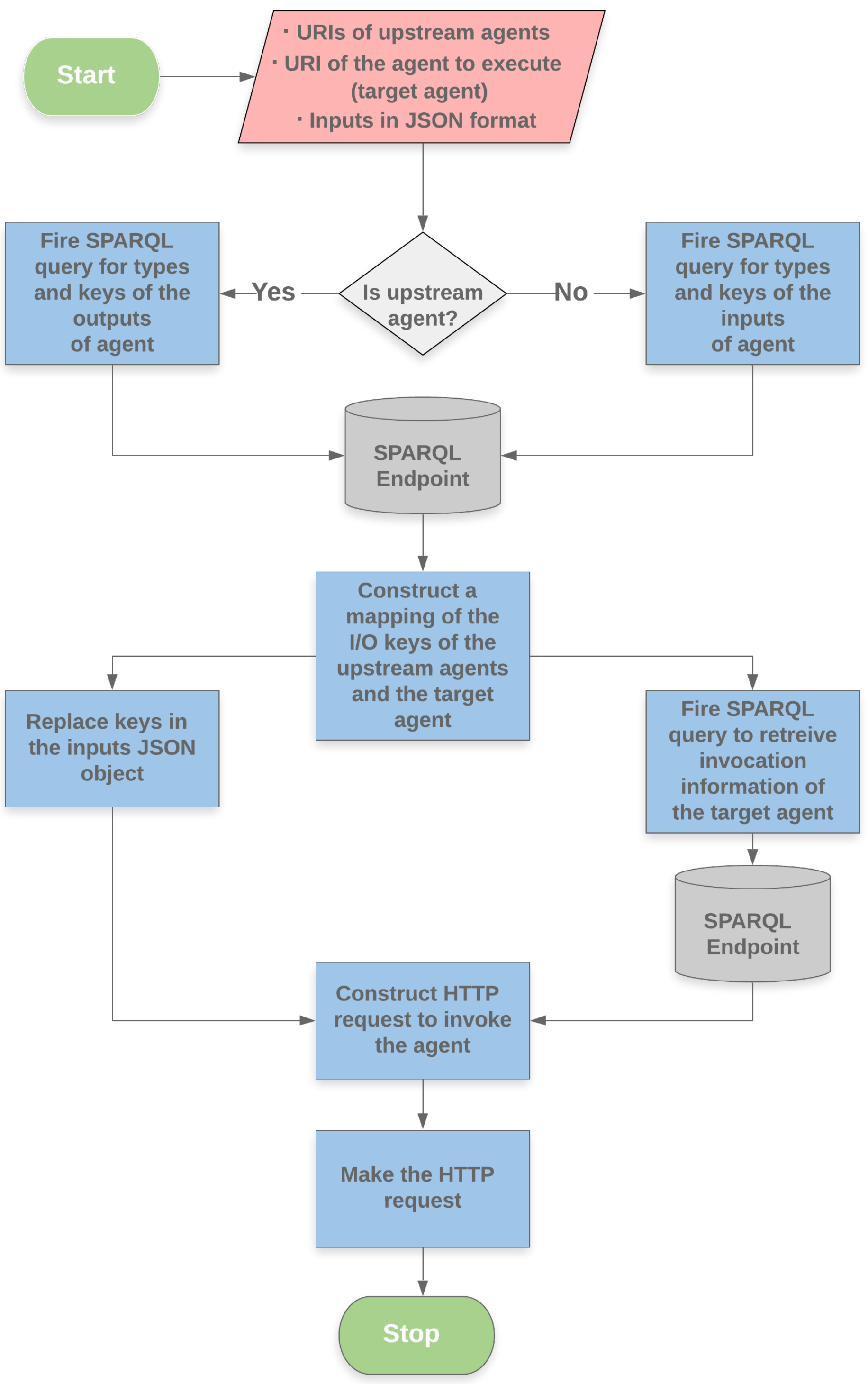
Role names	Domain and range restrictions on new roles
hasInvocation	$\exists$ ontoagent:hasInvocation.$\top \sqsubseteq$ msm:Operation $\top \sqsubseteq \forall$ ontoagent:hasInvocation.ontoagent:Invocation
hasHttpUrl	$\exists$ ontoagent:hasHttpUrl.$\top \sqsubseteq$ msm:Invocation $\top \sqsubseteq \forall$ ontoagent:hasHttpUrl. xsd:anyURI
hasKey	$\exists$ ontoagent:hasKey.$\top \sqsubseteq$ msm:Invocation $\top \sqsubseteq \forall$ ontoagent:hasHttpUrl. Datatypestring
isArray	$\exists$ ontoagent:isArray.$\top \sqsubseteq$ msm:MessagePart $\top \sqsubseteq \forall$ ontoagent:isArray. Datatypeboolean
hasType	$\exists$ ontoagent:hasType.$\top \sqsubseteq$ msm:MessagePart $\top \sqsubseteq \forall$ ontoagent:hasType. xsd:anyURI

A.5 Flowchart of agent discovery



Start
Collected inputs
Query inputs of all agents using SPARQL
SPARQL Endpoint
Mapping of agents and their input types
Go to next agent in the mapping
No agent left?
No
Yes
No
Push the URI to the result list
Yes
Are all the input requirements of the agent met?
Stop

A.6 Flowchart of agent execution


Start
· URIs of upstream agents
· URI of the agent to execute (target agent)
· Inputs in JSON format
Is upstream agent?
Yes
No
Fire SPARQL query for types and keys of the outputs of agent
Fire SPARQL query for types and keys of the inputs of agent
SPARQL Endpoint
Construct a mapping of the I/O keys of the upstream agents and the target agent
Replace keys in the inputs JSON object
Fire SPARQL query to retreive invocation information of the target agent
SPARQL Endpoint
Construct HTTP request to invoke the agent
Make the HTTP request
Stop

References

[1] H. Afshari, R. Farel, and Q. Peng. Improving the resilience of energy flow exchanges in eco-industrial parks: Optimization under uncertainty. *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering*, 3 (2):021002, 2017. doi:10.1115/1.4035729.

[2] M. Aiello, N. van Benthem, and E. el Khoury. Visualizing compositions of services from large repositories. In *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. IEEE, 2008. doi:10.1109/cecandeee.2008.149.

[3] E. Cimren, J. Fiksel, M. E. Posner, and K. Sikdar. Material flow optimization in by-product synergy networks. *Journal of Industrial Ecology*, 15(2):315–332, 2011. doi:10.1111/j.1530-9290.2010.00310.x.

[4] A. Eibeck, M. Q. Lim, and M. Kraft. J-Park Simulator: An ontology-based platform for cross-domain scenarios in process industry, 2019. Submitted for publication.

[5] F. Farazi, J. Akroyd, S. Mosbach, P. Buerger, D. Nurkowski, and M. Kraft. OntoKin: An ontology for chemical kinetic reaction mechanisms, 2019. Submitted for publication.

[6] D. Fensel, F. M. Facca, E. Simperl, and I. Toma. Web service modeling ontology. In *Semantic Web Services*, pages 107–129. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-19193-0_7.

[7] R. T. Fielding and R. N. Taylor. Architectural styles and the design of network-based software architectures. `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`. last accessed: 2019-04-13.

[8] K. Fujii and T. Suda. Semantics-based context-aware dynamic service composition. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):12:1–12:31, 2009. doi:10.1145/1516533.1516536.

[9] G. Gröger and L. Plümer. CityGML - interoperable semantic 3D city models. *ISPRS Journal of Photogrammetry and Remote Sensing*, 71:12 – 33, 2012. doi:10.1016/j.isprsjprs.2012.04.004.

[10] H. Haslenda and M. Jamaludin. Industry to industry by-products exchange network towards zero waste in palm oil refining processes. *Resources, Conservation and Recycling*, 55(7):713–718, 2011. doi:10.1016/j.resconrec.2011.02.004.

[11] P. Hennig and W.-T. Balke. Highly scalable web service composition using binary tree-based parallelization. In *2010 IEEE International Conference on Web Services*. IEEE, 2010. doi:10.1109/icws.2010.45.

[12] M. Klusch, A. Gerber, and M. Schmidt. Semantic web service composition planning with OWLS-XPlan. In *Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web*, pages 55–62. sn, 2005. URL https://www.aaai.org/Papers/Symposia/Fall/2005/FS-05-01/FS05-01-008.pdf.

[13] S. Kona, A. Bansal, M. B. Blake, and G. Gupta. Generalized semantics-based service composition. In *2008 IEEE International Conference on Web Services*. IEEE, 2008. doi:10.1109/icws.2008.118.

[14] J. Kopeckỳ and T. Vitvar. WSMO-Lite: Lowering the Semantic Web Services Barrier with Modular and Light-Weight Annotations. In *2008 IEEE International Conference on Semantic Computing*, pages 238–244, 2008. doi:10.1109/ICSC.2008.54.

[15] J. Kopeckỳ, T. Vitvar, C. Bournez, and J. Farrell. Semantic Annotations for WSDL and XML Schema. https://www.w3.org/TR/sawsdl/. last accessed: 2019-03-11.

[16] J. Kopeckỳ, K. Gomadam, and T. Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. In *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 619–625, 2008. doi:10.1109/wiiat.2008.379.

[17] Y. T. Leong, J.-Y. Lee, R. R. Tan, J. J. Foo, and I. M. L. Chew. Multi-objective optimization for resource network synthesis in eco-industrial parks using an integrated analytic hierarchy process. *Journal of Cleaner Production*, 143:1268–1283, 2017. doi:10.1016/j.jclepro.2016.11.147.

[18] Z. W. Liao, J. T. Wu, B. B. Jiang, J. D. Wang, and Y. R. Yang. Design methodology for flexible multiple plant water networks. *Industrial & Engineering Chemistry Research*, 46(14):4954–4963, 2007. doi:10.1021/ie061299i.

[19] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for web services. http://www.ai.sri.com/~daml/services/owl-s/1.2/overview/. last accessed: 2019-03-11.

[20] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eights International Conference on Principles of Knowledge Representation and Reasoning*, volume 2, pages 482–493, 2002. URL http://semanticweb2002.aifb.uni-karlsruhe.de/proceedings/Position/sheila.pdf.

[21] J. Morbach, A. Wiesner, and W. Marquardt. OntoCAPE: A (re) usable ontology for computer-aided process engineering. *Computers & Chemical Engineering*, 33(10): 1546–1556, 2009. doi:10.1016/j.compchemeng.2009.01.019.

[22] S. K. Nair, Y. Guo, U. Mukherjee, I. Karimi, and A. Elkamel. Shared and practical approach to conserve utilities in eco-industrial parks. *Computers & Chemical Engineering*, 93:221–233, 2016. doi:10.1016/j.compchemeng.2016.05.003.

[23] W. Nam, H. Kil, and D. Lee. Type-aware web service composition using boolean satisfiability solver. In *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. IEEE, 2008. doi:10.1109/cecandeee.2008.108.

[24] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20: 379–404, 2003. doi:10.1613/jair.1141.

[25] S.-C. Oh, D. Lee, and S. R. Kumara. Web Service Planner (WSPR). *International Journal of Web Services Research*, 4(1):1–22, 2007. doi:10.4018/jwsr.2007010101.

[26] A. M. Omer and A. Schill. Dependency based automatic service composition using directed graph. In *2009 Fifth International Conference on Next Generation Web Services Practices*. IEEE, 2009. doi:10.1109/nwesp.2009.20.

[27] M. Pan, J. Sikorski, C. A. Kastner, J. Akroyd, S. Mosbach, R. Lau, and M. Kraft. Applying Industry 4.0 to the Jurong Island Eco-industrial Park. *Energy Procedia*, 75:1536 – 1541, 2015. doi:10.1016/j.egypro.2015.07.313.

[28] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopeckỳ, and J. Domingue. iServe: a linked services publishing platform. In *Ontology Repositories and Editors for the Semantic Web Workshop at The 7th Extended Semantic Web*, volume 596, 2010. URL `http://oro.open.ac.uk/23093/`. last accessed: 2019-4-12.

[29] K. Raman, Y. Zhang, M. Panahi, and K.-J. Lin. Customizable business process composition with query optimization. In *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. IEEE, 2008. doi:10.1109/cecandeee.2008.152.

[30] P. Rodriguez-Mier, C. Pedrinaci, M. Lama, and M. Mucientes. An integrated semantic web service discovery and composition framework. *IEEE Transactions on Services Computing*, 9(4):537–550, 2016. doi:10.1109/tsc.2015.2402679.

[31] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218 – 238, 2014. doi:10.1016/j.ins.2014.04.054.

[32] M. M. Shiaa, J. O. Fladmark, and B. Thiell. An incremental graph-based approach to automatic service composition. In *2008 IEEE International Conference on Services Computing*. IEEE, 2008. doi:10.1109/scc.2008.141.

[33] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for Web Service composition using SHOP2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):377–396, 2004. doi:10.1016/j.websem.2004.06.005.

[34] R. R. Tan and K. B. Aviso. An inverse optimization approach to inducing resource conservation in eco-industrial parks. In *Computer Aided Chemical Engineering*, pages 775–779. Elsevier, 2012. doi:10.1016/b978-0-444-59507-2.50147-5.

[35] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. doi:10.1006/knac.1993.1008.

[36] B. T. C. Tiu and D. E. Cruz. An MILP model for optimizing water exchanges in eco-industrial parks considering water quality. *Resources, Conservation and Recycling*, 119:89–96, 2017. doi:10.1016/j.resconrec.2016.06.005.

[37] Y. Yan, B. Xu, and Z. Gu. Automatic service composition using AND/OR graph. In *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. IEEE, 2008. doi:10.1109/cecandeee.2008.124.

[38] C. Zhang, L. Zhou, P. Chhabra, S. S. Garud, K. Aditya, A. Romagnoli, G. Comodi, F. D. Magro, A. Meneghetti, and M. Kraft. A novel methodology for the design of waste heat recovery network in eco-industrial park using techno-economic analysis and multi-objective optimization. *Applied Energy*, 184:88 – 102, 2016. doi:10.1016/j.apenergy.2016.10.016.

[39] C. Zhang, L. Zhou, P. Chhabra, S. S. Garud, K. Aditya, A. Romagnoli, G. Comodi, F. D. Magro, A. Meneghetti, and M. Kraft. A novel methodology for the design of waste heat recovery network in eco-industrial park using techno-economic analysis and multi-objective optimization. *Applied Energy*, 184:88–102, 2016. doi:10.1016/j.apenergy.2016.10.016.

[40] L. Zhou, C. Zhang, I. A. Karimi, and M. Kraft. An ontology framework towards decentralized information management for eco-industrial parks. *Computers & Chemical Engineering*, 118:49–63, 2018. doi:10.1016/j.compchemeng.2018.07.010.