POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

SCoReS

SeNSori COmposition & REtrieval Services

Relatore:
Prof.ssa Letizia Tanca
Co-Relatori:
Prof. Pierluigi Plebani
Prof.ssa Miriana Mazuran

Tesi di laurea di:
Claudia Foglieni
Matricola 782875
Giovannni Meroni
Matricola 783419

Table of Contents

List of Figures

List of Tables

List of Algorithms

1 Sommario

La diffusione delle reti di sensori e la conseguente installazione all'interno di edifici residenziali, commerciali e industriali ha permesso di sviluppare soluzioni tecnologiche che spesso cadono sotto il nome di Smart Building. Tra queste soluzioni, i sistemi a supporto del risparmio energetico negli edifici stanno avendo particolare rilevanza.

L'approccio di base, che offre funzioni di controllo per mantenere bassi consumi, richiede che tutti i dati raccolti siano inviati ad un archivio centralizzato. Un amministratore deve poter esplorare tutti i dati e trovare la giusta interpretazione per operare al momento giusto.

Tra queste soluzioni vi è la piattaforma SeNSori: una piattaforma basata sui servizi adattivi per l'intelligenza ambientale il cui scopo è di monitorare il consumo riguardante tutte le utenze energetiche. Con due tipologie di dispositivi è possibile valutare ogni situazione: sensori, i quali registrano i dati riguardanti l'ambiente, ed attuatori, che sono utilizzati per compiere azioni di adattamento mirate a risolvere eventuali situazioni di criticità, o a migliorare il risparmio energetico.

Per nascondere l'eterogeneità e la complessità tecnica di tutti i dispositivi distribuiti in un edificio, un livello superiore sotto forma di servizi è stato fornito per interagire con i sensori. In questo modo, per ottenere i dati o controllare i sensori, è obbligatorio richiamare i metodi esposti da questi servizi.

Il presente lavoro di tesi punta ad arricchire gli strumenti messi a disposizione dalla piattaforma SeNSori attraverso le seguenti funzionalità:

- Una funzionalità di ricerca, arricchita con un algoritmo di similarità, al fine di trovare i servizi correlati.
- Una funzionalità di composizione, che consente all'utente di creare nuovi servizi combinando quelli esistenti.

Il nostro approccio ha portato alla creazione di una dashboard per l'amministrazione, che potrà essere utilizzata per sfruttare le nostre funzionalità principali: SCoReS (SeNSori COmposition & REtrieval Services).

2 Abstract

The diffusion of wireless sensor networks and the subsequent installation in residential, commercial and industrial buildings led to the development of technological solutions that often fall under the name Smart Building. Among these solutions, systems supporting energy saving in buildings are gaining greater importance.

The basic approach, which provides control functions for maintaining the energy consumption low, requires that all the gathered data are sent to a centralized repository. An administrator must be able to explore all the data and find the right interpretation in order to operate at the right time.

Among these solutions there is the SeNSori platform: an adaptive service-based platform for ambient intelligence whose aim is to monitor the energy consumption concerning all the energy utilities. It is possible to evaluate every situation by using two types of devices: sensors, that register measurements about the environment,and actuators, which are used to enact the adaptation actions that aim to solve possible critical situations or to improve the energy saving.

In order to hide the technical heterogeneity and complexity of all the devices deployed in a building, a service layer is provided to interact with the sensors. In this way, in order to obtain data or control the sensors, it is mandatory to invoke the methods exposed by these services.

The present work aims at enriching the tools provided by the SeNSori platform through the following features:

- A retrieval functionality, enriched with a similarity algorithm, in order to find relevant services.
- A composition functionality, which enable the user to create new services combining the existing ones.

Our approach has led to the creation of a governance dashboard that will be utilized by the administrator in order to exploit our core functionalities: SCoReS (SeNSori COmposition & REtrieval Services).

3 Introduction

In the recent years a new research area has been exploited by several researchers: the utilization of different sensors inside a building to monitor its energy efficiency. This idea generates the concept of Smart Buildings, as "green" buildings equipped with sensors that can analyze and interpret the received information in order to assess if the amount of consumed energy is adequate for all the activities. For this reason is necessary to combine the data transmitted by the building sensors with data about the context in which the building is located.

The basic approach provides efficient control functions for maintaining building condition (e.g., temperature, humidity, air cleaning, light control, and reducing unnecessary energy consumption). All the data are gathered by a network composed by different types of sensors, which send their measurements to a centralized repository. An administrator, that wants to analyze all the received information, needs to explore all the data and find the right interpretation in order to operate at the right time. This procedure is time consuming and presumes several bases:

- A general idea of the layout of the buildings.
- The knowledge about the different types of sensors present in the network.
- An understanding about the measurements, their values and their associated dates.

Our solution to this problem is an approach based on **SeNSori**: an adaptive service-based platform for ambient intelligence whose aim is to monitor the energy consumption concerning all the energy utilities, i.e., water, electricity, gas consumption in one or more buildings.

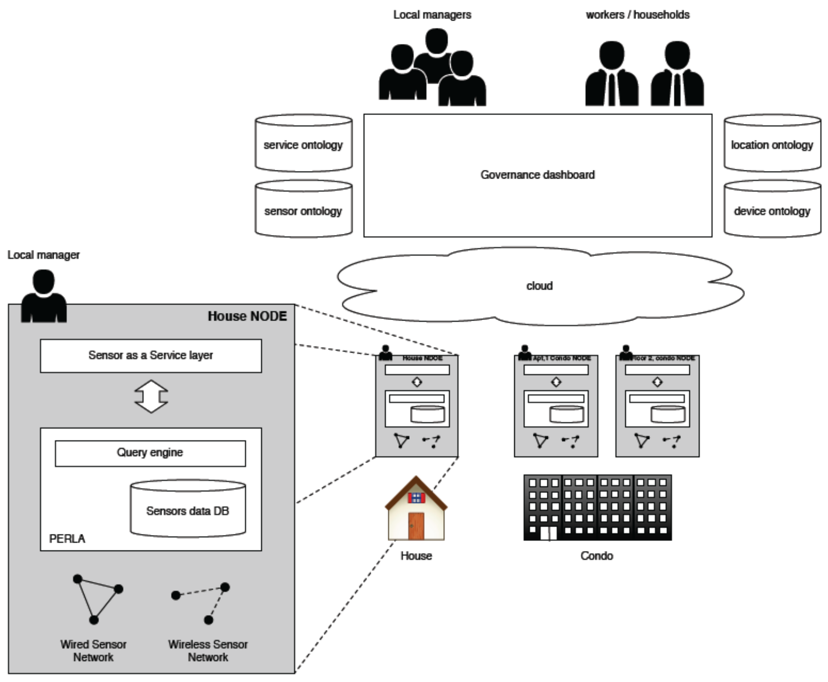


Figure 1: SeNSori architecture

Inside the SeNSori network there are two types of devices:

- The sensors, which register measurements about the environment,
- The actuators, which are used to enact the adaptation actions that aim to solve possible critical situations or to improve the energy saving.

The user does not directly interact with this network, but a service layer is provided in order to hide the technical heterogeneity and complexity of all the devices deployed in a building. In this way, in order to obtain data or control the sensors, the administrator needs to invoke the methods exposed by these services.

We have selected two main area to study in deep, in order to offer the maximum support to the administrator or the common user. For this reason the focal points of this thesis are:

- A search functionality, enriched with a similarity algorithm in order to find relevant services;
- A composition functionality, which enable the user to create new services combining the existing ones.

Our approach has produced on a governance dashboard that will be utilized by the administrator in order to exploit our core functionalities: **SCoReS** (SeNSori COmposition & REtrieval Services).

As stated by the name of our project, we offer two principal modules:

- A retrieval functionality, to search information about a service, specifying a set of parameters.
- A composition functionality, to create new services.

The retrieval module of SCoReS is based on a similarity algorithm, which analyzes the query submitted by the user and finds the services that match the user's query. The final list is ordered from the most relevant result to the least one, with a complete analysis that compares all the main aspects characterizing a service: the type of sensors, the most recent data and its location.

The composition module instead offers the possibility to create new services that can satisfy new needs that can raise during SeNSori's usage. This module can produce many composed services covering the most disparate needs: from a new service that implements an automatic action on an actuator based on data collected by a sensor, to a weekly report about the energy consumption.

As surplus value inside our composition module, we provide the possibility to save composite service templates, in order to create a model that can be re-utilized whenever a user needs to create a service similar to the previous ones. In this thesis, we will focus on the development of SCoReS. Our prototype is based on a set of data registered in the past, with a complete ontology that covers all the possible functionalities of our system. The programmer can utilize this standard configuration or personalize it.

The rest of the thesis is organized as follows.
Section 4 analyzes the existing solutions, with the advantages and disadvantages of their approaches.
Section 5 presents the basic tools and technologies for our project.
Section 6 discusses the SCoReS architecture, whit a general overview of all its modules.
Section 7 presents a detailed description about our similarity algorithm.
Section 8 is used to describe the module about the service composition.
Section 9 describes the entire implementation of the core functionalities is presente.
Section 10 concludes the thesis mentioning what has been done and proposed future works.

4 State of art

Our system is composed by different technologies. In order to understand and utilize the right tools for every component, we have considered several fields of interest:

- Algorithms evaluating the degree of similarity of web services.
- Tools to create composite services.
- A protocol to manage the communication between the graphical interface and our system.

These three areas represent the core functionalities of SCoReS. In fact our system is composed of two main modules, which interact with the user using our protocol.

As starting point we have analyzed the existing models and architectures that exploit the concept of sensor networks. As described by [1] there are two important factors in every system:

- The design of the network itself, with all the characteristics supported by the sensor,
- The communication architecture and the algorithms and protocols developed for each layer.

In similar works, sensor networks are used to monitor energy consumption in several domains, such as extending the functionalities of sensors managed through NAGIOS [8] to monitor the energy consumption and, by exploiting the functionalities of a service oriented architecture, improving the energy saving by optimizing the resource usage. The same approach is utilized in SCoReS, though the different sensors and actuators are not directly accessible, and can be used only by interacting with a service layer that allows only certain operations.

A much more similar architecture is the one presented by [30], where a coherent infrastructure is developed, which treat sensors in an interoperable, platform-independent and uniform way. This work addresses the problems related to the format of the information exchanged by the sensor and user. The solution proposed is similar to the one implemented in SCoReS, with the adoption of the Representational State Transfer (REST) web services concept and the usage of JavaScript Object Notation (JSON), as a lightweight approach that simplifies the messages.

A different architecture, based on a multi-agent system, is proposed in [22], which adopts machine learning techniques to predict the occupancy of rooms and, consequently, to automatically adapt the configuration of HVAC systems, but it limits the applicability to a specific domain. With SCoReS the central figure that operate the final decision is the administrator, a human being supported by knowledge built on top of a set of information collected and properly analyzed by our system.

Fabric [14], a middleware inspired by the SOA paradigm to manage sensor networks that follows an approach similar, differs from SCoReS because is more process-oriented to better manage the events that might arise from the sensors. Our project is more data-oriented as the main objective is to analyze the data in order to retrieve or create new services.

A much more similar approach to SCoReS is proposed by the EU project SANY (Sensors ANYwhere) [2], that introduces SOA for accessing and managing sensors regardless of their technical aspects. The main difference is that there are no ontologies involved, which in our approach assume an important part in the execution of the service similarity algorithm.

A more detailed survey about interoperability, integration, overhead, and bandwidth is described by [15], which highlights the advantages of an IP-based and RESTful architecture approach as the most suitable solution. The purpose remarked is the possibility to enable interaction of users with the sensor network in the same way as with any website while ensuring energy efficiency.

Considering the correlation between the architecture and the data gathered, TinyRest [20] is the solution that provides a set of functionalities similar to SCoReS. In fact the described work exploits REST principles by defining a mapping between HTTP REST messages and TinyOS messages in order to allow high level control and providing a software layer to enable HTTP data exchanges between the middleware and the devices. The same approach is implemented in SCoReS with the retriever module.

The idea of using ontologies to register and discover sensor services has already been introduced in [17] and [4]. Differently from these approaches, in SCoReS we have enriched the already known OWL-S ontology for the representation of services and used standard data models and languages. With the creation of new services by using our composition modules, the administrators can add new definitions in the ontology, with the same data models.

4.1 Similarity's algorithms

In SCoReS we need to compare different services starting from a set of parameters chosen by the user. The result is an ordered list with all the pertinent services.

In order to rate our set of services from zero, complete mismatch with all parameters selected by the user and the service's ones, to one, perfect match, we have analyzed different approaches. During our research we have discover two main fields of applications:

- Algorithms for similarity based on semantic data.
- Algorithms for similarity based on numeric or measured data.

In fact we need a method that considers both the request submitted by the user and the real-time data registered by every service. With this information SCoReS can search the relevant services that can satisfy that request. [7] present an interesting idea, about similarity calculated on the output of every service. This work is one of our starting point, even if no similarity function is explicitly mentioned.

The basic idea that we have developed in our project follows the guidelines described by [16] and [5]:

- We have five classes of services that are mutually exclusive.
- From the information present in our system we can classify every service in one of these classes, with respect to the request sent by the user.
- Every class has a corresponding set of penalties, which creates the final ranking.
- Only the sub-set of services that has a ranking higher that a certain threshold will be presented to the user.

The final score for every service will be composed by two different sections, as described by [11]. To determine the similarity of a request to a particular Web service, we compute:

- Structural Similarity, that in our case is computed from the output of every service,
- Semantic Similarity, calculated by the data present in the ontology.

For the structural similarity we can't use the approach described by [23] and [37], because it supposes that in our system we store the results of every single query, in order to re-use the similarity computed in the past to refine the future ranking. But our services are not a static entity; they can change during time, with new values registered by the associated sensors. The ranking calculated in the past could not be the same in the future.

Also the approach proposed by [33] is too much overkill for our system. We have used the idea of different scoring functions for our different types of services. In this way we can select the best method to evaluate a particular service, starting from its definition stored in the ontology. These functions have a different flow of operations, but they all have these comparable aspects:

- The request of the data registered by the associated sensors.
- Every datum is composed by two component, a value and the corresponding timestamp for the registration date.
- The result is a penalty, with a value between zero and one.

We have discarded [12], [19] because their approach is based on the semantic. In our system we don't need a function to calculate semantic similarity. In fact we use the ontology in order to understand if two different concepts are similar or not.

4.2 Service composition

In order to exploit our service composition functionality, we have analyzed the existing solutions. With the consideration stated in [6] and [21], the existing models and languages are very complex and offer different tools to cover all the possible aspects. In our case we need a flexible system, with standard modules that the user can put together to build his personal workflow. For this reason we have discarded the languages that do not offer this flexibility.

Also solutions created for the REST architecture, like the one described by [38], don't cover the model that we need. In fact in our model the user can insert a code snippet, a function written in JavaScript. In order to evaluate this function and point out eventual errors, we need to add extra controls, that are not supported.

A good starting point is the WS-BPEL, with the functionalities exposed in [25]. We have modeled our system to reflect a portion of the features offered by BPEL, tuning our workflow engine to support flexibility and personalization. We have adopted this strategy because the implementation from scratch of our ad-hoc system offers the possibility to add future modules without being subject to a steep learning curve, as opposed to BPEL.

4.3 Other technologies: communication's protocol

There are several protocols that can be chosen for modeling the communication between different systems. Since our project is web-based, we have analyzed the following options:

- REST, an architectural style for distributed system [35].
- SOAP, a protocol specification for exchanging structured information [36].

As analyzed in [29], these two approaches have different strengths and weaknesses.

For SOAP we have as advantage the transparency and the independence. In fact the same message in the same format can be transported across a variety of middleware systems. Using WSDL to describe a service interface helps to abstract from the underlying communication protocol and serialization details as well as from the service implementation platform. Further advantage of WSDL contracts is that they provide a machine-processable description of the syntax and structure of the corresponding request and response messages and define a flexible evolution path for the service. In particular, WSDL can model service interfaces for systems based on synchronous and asynchronous interaction patterns. But SOAP has also some weaknesses: it is important to avoid leakage across abstraction levels, interoperability problems can occur when, for instance, native data types and language constructs of the service implementation are present in its interface and impedance mismatch between XML and existing (object-oriented) programming languages.

REST services are perceived to be simple because REST leverages existing well-known W3C/IETF standards (HTTP, XML, URI, MIME) and the necessary infrastructure has already become pervasive. Such lightweight infrastructure, where services can be built with minimal tooling, is inexpensive to acquire and thus has a very low barrier for adoption. On the operational side, it is known how to scale a stateless RESTful Web service to serve a very large number of clients, thanks to the support for caching, clustering and load balancing built into REST. Due to the possibility of choosing lightweight message formats, REST also gives more leeway to optimize the performance of a Web service. But there is some confusion regarding the commonly accepted best practices for building RESTful Web services. Another limitation makes it impossible to strictly follow the GET vs. POST rule. For idempotent requests having large amounts of input data (more than 4 KB in most current implementations) it is not possible to encode such data in the resource URI, as the server will reject such "malformed" URIs6 – or in the worst case it will crash, exposing the service to buffer overflow attacks. The size of the request notwithstanding, it may also be challenging to encode complex data structures into a URI as there is no commonly accepted marshalling mechanism. Inherently, the POST method does not suffer from such limitations.

Analyzing the requirements of our system, we have decided to implement REST architecture with JSON as message format. JSON is a text-based open standard designed for human-readable data interchange [34]. Our architecture respects the principles exposed by [28] and [13], with the base assumption that a service essentially is a function-oriented interface exposed through distributed object technology.

We have opted to not implement any framework, as described in [39], in order to maintain agility and freedom in our message format, adding only the resources need by our system. For the same reason we have discarded the model proposed by [32], [24] and [3].

The REST model proposed by [13] can be good for our approach. In fact the authors suggest that the mentioned REST model is conceptually similar to the semantic web standard for the SPARQL language. With this starting point we can model all the concepts present in our ontology as REST resources. Also one of the main goals of the formalization of RESTful semantic services is the possibility of describing interaction patterns between resources and clients, simplifying our interaction with the graphical interface. Only the action with a REST definition can be exploited by the user.

Also [27] offers an interesting idea: a composite RESTful service is a special kind of intermediate element which not simply forwards requests to upstream origin servers but may decompose a request so that it can be serviced by invoking more than one origin server. In our case the request is sent by the graphical interface to one of our modules, which can interact with the other ones in order to compose the final message.

For a better compatibility between different modules, we have opted to follow the guidelines suggested by [9]. In our case every function important for the user has an equivalent REST end-point. It is easy to add a new function or modify the existing ones, because our architecture reflects the real data of our system.

5 Tools and technologies utilized

5.1 Ontologies

An ontology is a structure that formally represents knowledge as a set of concepts and relationships between them within a domain. An ontology provides a shared vocabulary, which can be used to model a domain, that is, the type of objects and/or concepts that exist, and their properties and relations.

Aims of ontologies are describing the knowledge in a computable form, standardizing and providing a rigorous definition for the terminology used in the domain, and allowing automatic classification and inference. An ontology is composed by the following components:

- Individuals: instances of objects.
- Classes: concepts, sets and types of objects.
- Attributes: properties that objects and classes can have.
- Relations: ways in which classes and individuals can be related to one another.

5.2 OWL

OWL (Web Ontology Language) is a family of knowledge representation languages designed for use by applications that need to process the content of information instead of just presenting information to humans. The languages are characterized by formal semantics and RDF/XML-based serializations

5.3 RDF

RDF (Resource Description Framework) is a family of specifications originally designed as a metadata data model that are mainly used for conceptual description or modeling of information.

5.4 SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) is a query language for ontologies which is able to retrieve and manipulate data stored in RDF format. More in detail, with SPARQL it is possible to pull values from structured and semi structured data represented in RDF, explore RDF data by querying unknown relationships, apply transformations to RDF data, and join multiple

RDF repositories in a single query.

SPARQL provides the following query variations:

- SELECT, which is used to extract values in a tabular form.
- CONSTRUCT, that can be used to extract information and transform the results into valid RDF.
- ASK, which answers the query with a simple true/false response.
- DESCRIBE, that returns the RDF graph.

5.5 Apache Jena

Jena is an open source Semantic Web framework for Java, which provides an API to extract data from and write to RDF graphs, and also allows the execution of SPARQL queries.

5.6 REST

REST (REpresentational State Transfer) is an architectural style for building large-scale distributed hypermedia systems and, in conjunction with the HTTP protocol, is often used for implementing web services.

The REST architectural style is based on four principles:

- Resource identification through URI: a RESTful Web service exposes a set of resources which identify the targets of the interaction with its clients, which are identified by URIs.
- Uniform interface: resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE.
- Self-descriptive messages: resources are decoupled from their representation so that their content can be accessed in a variety of formats (e.g., HTML, XML, plain text, etc.).
- Stateful interactions through hyperlinks: every interaction with a resource is stateless, i.e., request messages are self-contained, and stateful interactions are based on the concept of explicit state transfer.

5.7 JSON

JSON (JavaScript Object Notation) is a text-based open standard designed for human-readable data interchange. It is often used for serializing and transmitting structured data over a network connection.

6 General Architecture

Our system is composed of two different modules, which represent its core features:

- The service retriever, which allows the user to search particular services.
- The service composer, that lets the user create new services.

Both modules are based on different data sources, which enrich our system with specific information:

- An ontology, with semantic data.
- A set of services, divided into simple and composite ones.

Figure 2 shows the general architecture of SCoReS, with its main elements.

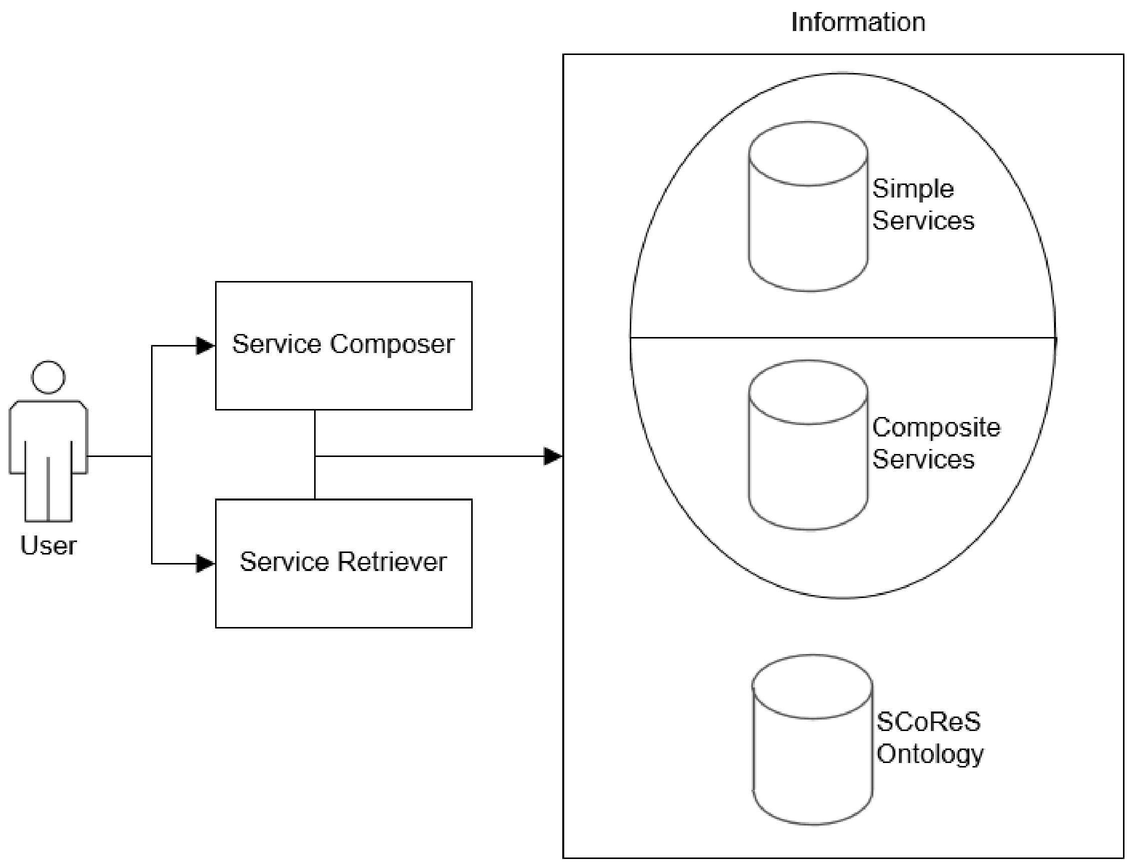


Figure 2: General architecture of SCoReS

The two modules are independent from each other. The user can interact with the graphical interface, which chooses the right module to forward the request. In this way the system can be extended with new modules, without modifying the existing ones.

6.1 Information's types present in the system

A service is a set of operations that the user can apply to the associated sensor, in order to gather real time data and analytic information.

Our system is based on this definition in order to communicate with the user.

There are two different types of services:

- The simple ones, which offer a single operation on a single sensor.
- The composite ones, whose operations are a workflow composed by single services and/or other composite services.

Both kinds of services have a set of attributes, which provides information about the service's main characteristics. If the user wants more specific information, he has to query our system for low level information.

The composite services can be created at runtime by the user, using pre-existing template or creating a new pattern for the operations' flow. Every new service is indexed and made available for future search or execution.

All the information can be extracted from three different sources:

- An ontology, that specifies the meaning for every concept.
- A database with real-time data registered by every sensor.
- A database with the definition of every composite service.

6.1.1 Ontology

The ontology is made up of four different sections:

- A section with the definition of services and their direct attributes.
- A sensor ontology, with details about sensors and actuators active in our system.

- A device ontology, with information about the type and energy consumption.
- A location ontology, with the data about the spatial environment.

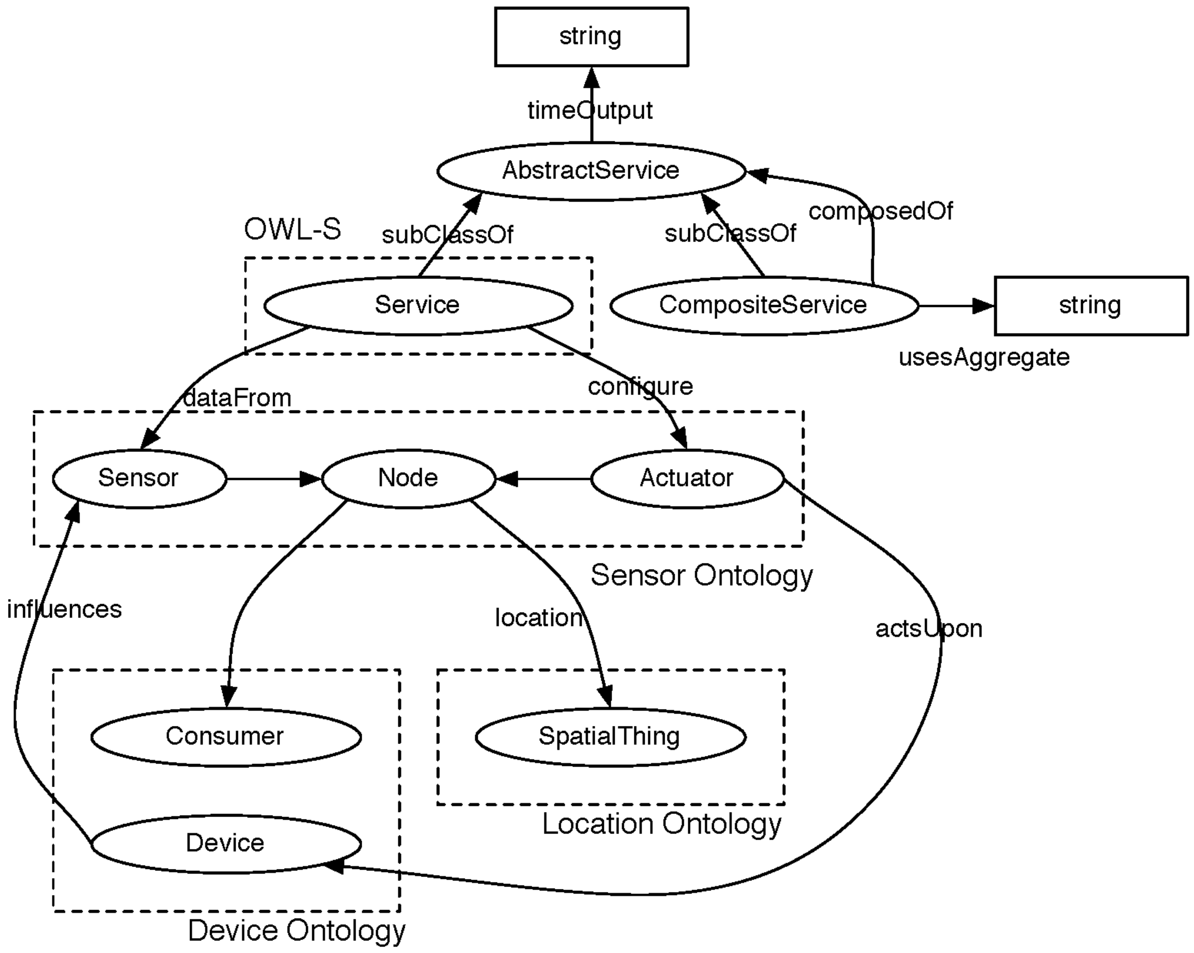


Figure 3: Structure of ontology, with link between different concepts

The different sections are linked one to another, creating a set of well defined paths that the system can navigate in order to connect different concepts. For example it is possible to find where a particular service is located by following this path:

- From the Service entity we can connect to the corresponding sensor with the relationship dataFrom.
- A Sensor is a specialization of the Node concept in our network,

- From our corresponding Node Entity is possible to find the related Location entity with the relationship location.

The ontology allows our system to establish a hierarchy between different concepts. For example, inside the definition of location, we can find a set of attributes like room, floor and building. With the ontology we can model the hierarchy like this:

- A building is composed of different floors.
- A floor is composed of different rooms.
- A room can house different locations.

The user can understand without difficulties how the location is structured and we can use every step in our hierarchy to recognize where a particular sensor is located.

The ontology also offers the possibility to evaluate the similarity between different concepts. For example, two different sensors in two adjacent positions can be similar if they register a temperature value during the same time span. The two measured temperature values have the same units of measurements, in our case Celsius Degree, and therefore can be compared.

6.1.2 Database with real-time data

The database with real-time data is used to store new information registered by the sensors and to analyze previously collected data.

The user can access all this information by using the operations provided by the service associated to the requested sensor. If he needs aggregated results or he wants to combine different data in order to perform a new analysis, the user can create a new service.

6.1.3 Database for composite services

The information about the composite services is saved in the second database. In this way we can catalog all the data concerning aggregated services, distinguishing two different contributions:

- The templates, a set of models that the user can re-utilize in order to create new services.
- The definition of new services.

Inside these definitions we store all the information that our system needs in order to be able to run composite services.

As additional data, every composed service is linked to its original template. In this way the user can know how a template is implemented and the differences between composite services originated from the same template.

6.2 Service Retriever

Our system offers a feature to search and explore our complete set of services, allowing the user to find the right information for its query.

There are two possible ways to compose a query:

- Creating a request by specifying a list of standard parameters,
- Writing a request in SPARQL.

The standard parameters are a set of data, which the user can select in order to filter services according to his needs. For a request the user doesn't need to know a priori the value for every parameter. Our system offers a series of methods to search and discover every possible value for the set of parameters. In this way the user can select the right values for his request, like the location of interest or the type of real-time data. This method is similar to a wizard-like procedure, where in every step is possible to choose if a parameter is important or not and which value is right for the final result.

In this first case there are two possible interpretation of a query:

- A strict one, where the result set must satisfy every single parameter.
- A relaxed one, where the user can relax some constrains, in order to include some services that would be excluded from the strict interpretation.

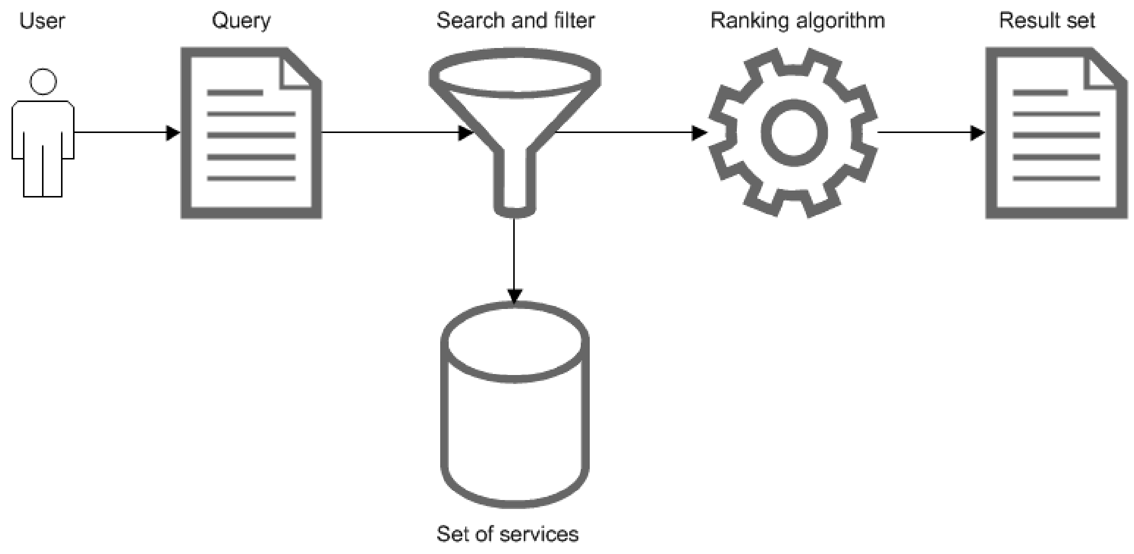


Figure 4: General features of service retriever

If the user wants to create a more complex query, he can use the SPARQL language in order to request particular services. In this case he can directly interact with the ontology and he has wider options of parameters. This is an advanced feature, that only expert user can utilize. In fact in order to exploit this method it is necessary a good knowledge of different concepts present in the ontology and the ability to write a query in SPARQL.

The result set of the relaxed interpretation will include every service present in the strict one, plus other services that are compatible to the non-relaxed parameters. For example: if the user requests services for temperature on floor 22, but he also want to loosen up the constrain about the location, our system will select all the services on floor 22, but also the ones present in other floors. For this partial result only the ones that register a temperature will be shown in the result set.

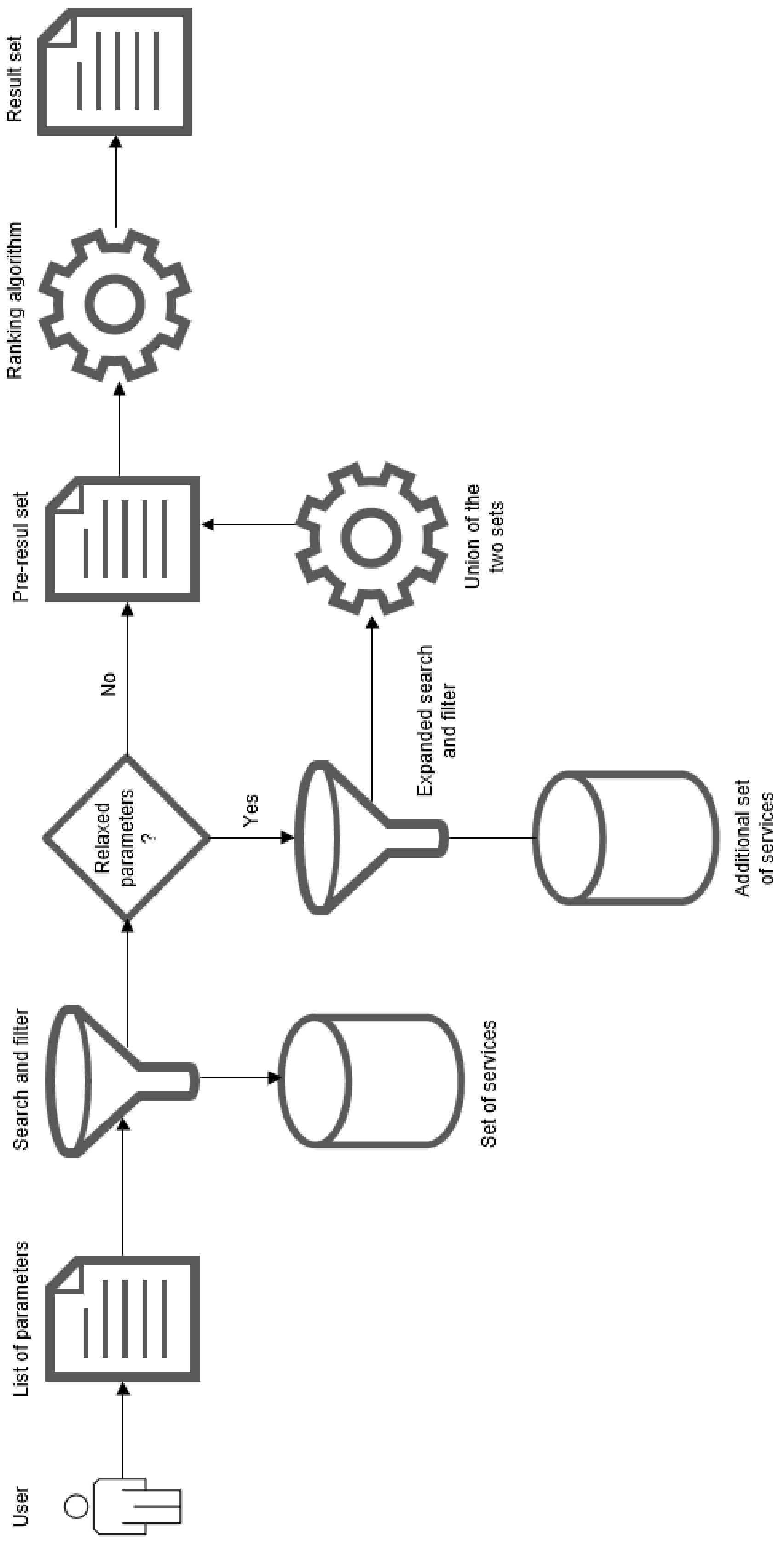


Figure 5: Strict vs relaxed queries

For every query, the user can get the following results: an ordered list containing information about simple and composite services matching the query requirements, or an error. The two results' types are mutually exclusive.

A result list is composed by one or more services and it is ordered from the most pertinent to the least one. In section 7 we will describe the analytical procedure to select and order relevant services.

The result list will include only simple services. If the user wants to retrieve composite services, he can invoke a set of methods that analyze that type of services. In fact the composite ones are structured and have a behavior completely different from the simple ones. Comparing the two types is computationally expensive: in order to understand if two services have a similar output, our system would need to execute every step in the composite service. Instead for a user is much simpler to understand which type can better fulfill his requirements.

An error will be displayed when the request sent by the user is not well formatted or our system doesn't have any data related to the submitted query. In this case the user can choose to re-create his query, relaxing some constraints or removing some parameters.

6.3 Service Composer

In order to create new services, the user can choose between two solutions:

- Load a pre-defined template.
- Create a new template, defining the general queries for searching the right services.

After the user completes the second solution, all the information inserted in the template are stored in our system. In this way future users can access this new template in order to generate new composite services.

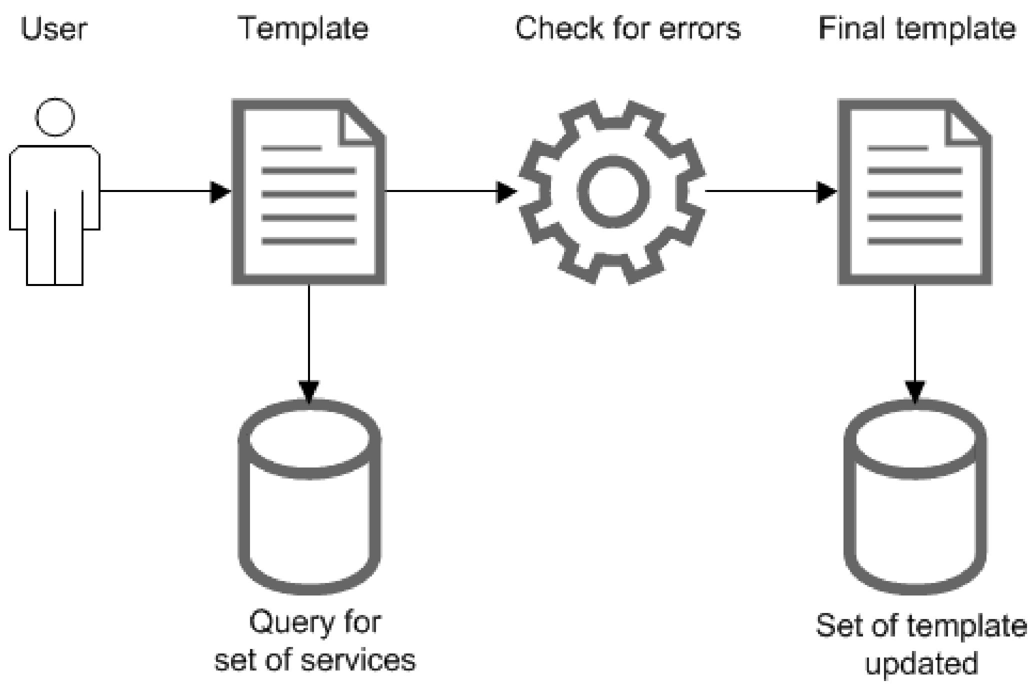


Figure 6: Operations to save a new template

In both cases the final template is composed by:

- A series of standard operations, like branch, loop or sequence.
- A query whenever a service should be invoked, that defines the characteristics of the service that the system will utilize in future.
- An optional transformation, like a function to search the maximum value.
- A result output, which can be a standard type or a report.

For the optional transformation, the user can choose from:

- A standard list, that contains a series of pre-defined operations, like a maximum, a minimum and a mean function.
- Insert a code fragment written by the user. This second solution is called snippet. The language for snippets is JavaScript and our system will validate every code's portion. If there are some errors, the user will be notified by the system.

The final output is chosen by the user. There are different types:

- A value computed dynamically during the workflow execution.
- The output of a particular service, which can be dynamically selected during the workflow execution.
- A report about the result of an operation, like the activation of an actuator.

After the user has defined his template, he can select every single service for every invoker's operation. The invoked services can be simple or composite. If all selected services are correct, namely they exist and are obtained from the original queries present in the template, the user can save the new composite service.

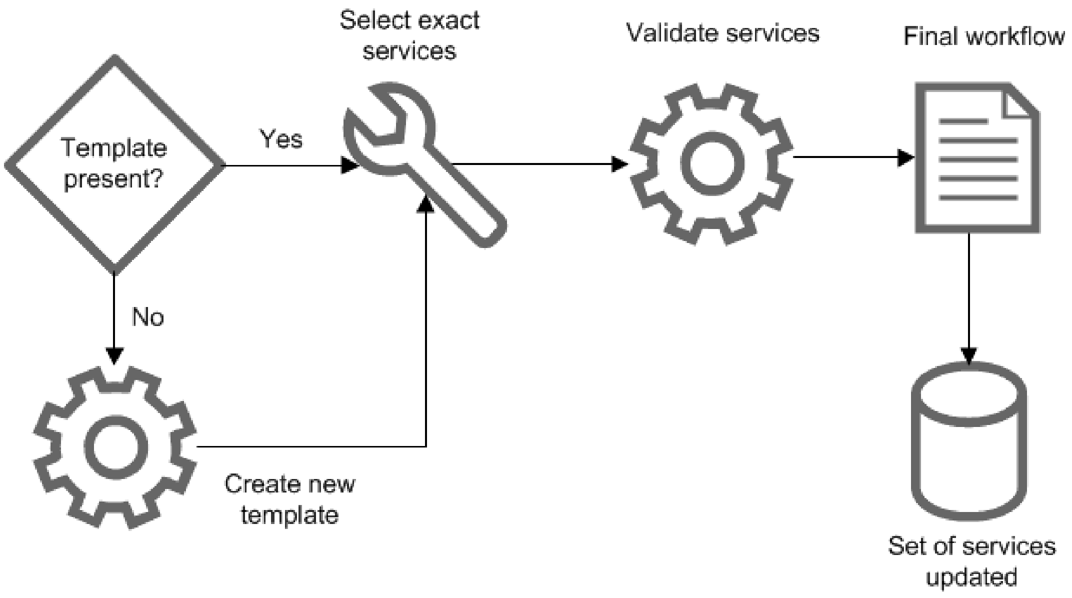


Figure 7: Operations to save a new composite service

This is the standard way to save a new executable composite service. In this way our system can run the new configuration and add it in the search list. In future this service will be available to be executed also by other users.

From the same template it is possible to create different composite services. In fact for every query presents in the template, the user can select different services present in the result set, customizing and tuning the final workflow. Our system offers a feature to request the list of composite services created from the same template, helping the user finding similar workflows.

7 Service similarity algorithm

A very important part of SCoReS service retrieval component is the service similarity algorithm, which is responsible for returning only meaningful results whenever the user performs a query concerning services. The service retrieval process is composed by the following two steps: at first all services that exactly match the user query are retrieved, and then the service similarity algorithm, which filters out results that are not interesting and adds other ones that are worth including, is invoked. More in detail, services are considered not interesting when their returned data have poor quality, whereas they are worth including when they don't precisely answer the user's query but the returned data have good quality.

Most of the research efforts in the service similarity topic concerns the comparison of generic services, and thus service similarity is evaluated by syntactically and semantically analyzing the service interface, and all of its parts, such as service description, methods' and parameters' names, and input and output data types. Since SCoReS simple service interfaces are predefined and only three kinds of services exist (*Last One*, *Last X* and *Interval*), this approach would not be much useful in our scenario.

What would be useful is the ability to compare services according to their input and output values. However, very few research articles concerning this approach have been written, probably due to the specificity of that problem: in [7] the problem of service similarity is dealt by clustering services that, for the same set of input values within the service's domain, return similar output values. However, this article does not mention how the service similarity function could be, but only says that should return values in the [0-1] range, with 1 for perfect match and 0 for complete mismatch.

In order to compute the degree of matching for a given pair of service and request, we have found that the approaches described in [16], [11] and [5] can be useful: every found service can be classified in five different categories.

These categories are:

- **Exact match**: service S exactly matches request R.
- **Plug-in match**: when the Service S provides at least the required functionalities and possibly adds new ones.
- **Subsume match**: when the functionalities provided by the service S are less than the required ones.

- **Intersection match**: when the request R and the service S present some common functionalities.
- **Mismatch**: when no common functionalities exist between the request R and the service S.

For SCoReS the first three categories are the best to offer to the user. In fact, from the request point of view, the first two kinds of match can be considered equivalent, since in both cases the offer fulfills the request, whereas in the case of subsume and intersection match the offer satisfies only partially the request.

There are two important characteristics, described by [11], that SCoReS must calibrate:

- **Granularity**: The results of the matchmaking are coarse-grained. That is, the matching services are associated only with some general similarity degrees (exact, plug-in, and so on) and we cannot further discriminate between services that have the same similarity degree.
- **Precision**: The matchmaking algorithm should provide high precision; meaning that those Web services that are actually labeled as matching should be compatible with the requests.

Due to the fact that we couldn't find any solution that fully solved the problem of evaluating service similarity based on input and output values, we chose to design and implement from scratch our service similarity algorithm. In fact most of the algorithms consider as a prominent part the analysis of the meaning's similarity, but in our case the real data need to be the relevant part. We can utilize the ontology in order to understand the meaning of every concept and to correlate different terms.

More in detail:

- SCoReS similarity algorithm at first retrieves all services that exactly answer the user's query (see Algorithm 1 line 1).
- If specified by the user, it adds to the result list services whose associated sensors are not too far from the location specified in the initial query (see Algorithm 1 lines 2 and 2).
- If specified by the user, it adds services having a time output property different from the one specified in the initial query (see Algorithm 1 lines 3 and 3).

- After that, a score of 1 (perfect match) is assigned to each previously found service (see Algorithm 4 lines 1 to 4).
- SCoReS similarity algorithm then computes the actual meaningfulness by multiplying the service's score by a penalty which is function of the quality of the values returned by the service (**service output penalty**, see Algorithm 4 lines 6, 7 and 8). If a service is unreachable the algorithm is unable to get any result from this one, for this reason the service is removed from the results list.
- Then the system multiplies each service's score by a penalty which is function of the distance between the associated sensor and the one specified in the initial query (**location penalty**, see Algorithm 4 lines 9 and 10).
- After the previous steps SCoReS similarity algorithm multiplies each service's score by a penalty based on the degree of compatibility between the service's time output property and the one specified in the query (**time output penalty**, see Algorithm 4 lines 11 and 12). The time output property reflects the kind of every service.
- Finally returns to the user only services whose score is above a certain threshold, and sorts them in descending order by their score (see Algorithm 1 lines 5 to 11).

Table 1 summarizes the penalties computed by SCoReS similarity algorithm in order to evaluate service similarity.

Penalty	**Meaning**	**Can be optional?**
Service output penalty	It is based on the real data registered by the sensor associated to the service.	No, it is computed for every query.
Location penalty	It is based on the position of every service and its relative distance from the original location requested by the user.	Yes, the user can choose to extend or not his query to different locations.
Time output penalty	It is based on the service's kind and how it is different from the original request.	Yes, the user can choose to extend or not his query to different kinds of services.

Table 1: Different penalties in SCoReS service similarity algorithm

A general complete execution follows these steps:

Algorithm 1 General operations for similarity algorithm

```
1: serviceList = retrieveServices(query parameters, ontology)
2: serviceList = serviceList U findNeighborServices(query parameters, ontology)
3: serviceList = serviceList U findTimeOutputCompatibleServices(query parameters, ontology)
4: rating = rankServices(serviceList, query parameters)
5: serviceList.sortDescending(rating, descending)
6: rating.sortDescending()
7: for all i = 0: serviceList.length do
8:     if rating[i] ¡ threshold then
9:         serviceList(i).remove
10:    end if
11: end for
12: return serviceList
```

Algorithm 2 Find neighbor services function

```
if query parameters.neighbor extension = true then
    query parameters.remove(location)
    return retrieveServices(query parameters, ontology)
else
    return null
end if
```

Algorithm 3 Find time output compatible services function

```
if time output extension = true then
    query parameters.remove(time output)
    return retrieveServices(query parameters, ontology)
else
    return null
end if
```

Algorithm 4 Rank service function

```
Double[] rating = new Double[serviceList.length]
for all k = 1: serviceList.length do
    rating(k) = 1
end for
for all k = 1: serviceList.length do
    service data = invokeService(serviceList(k))
    SOPenalty = computeServiceOutputPenalty(service data, query parameters.data constraints)
    rating(k) = rating(k) * SOPenalty
    LPenalty = computeLocationPenalty(serviceList(k), query parameters.location)
    rating(k) = rating(k) * LOPenalty
    TOPenalty = computeTimeOutputPenalty(services(k), query parameters.time output)
    rating(k) = rating(k) * TOPenalty
end for
return rating
```

Since service meaningfulness must be in the range [0-1], penalties are applied by multiplying by the location penalty (LPenalty) and the time output penalty (TOPenalty) the service output penalty (SOPenalty), all in the range [0-1], according to the following formula:
*Similarity = Similarity * SOPenalty * LPenalty * TOPenalty.*

7.1 Support data preprocessing

In order to compute the similarity between different services, we need to gather some data from different sources, since information required for the algorithm is partially distributed in the ontology and in the data returned by the services:

- Firstly we query the ontology for services having node type, kind, category, time output and/or location specified in the user query (if the user asked for neighbors extension or not strict time output, the time output and the location constraints are removed).

- Having done so, for each result we store service name, time output and associated node.
- If neighbors extension was asked by the user, fine-grained information about the node location are extracted from the ontology, like the room, floor, building and campus where the node is, in order to be able to compute the location penalty.
- After that, we invoke each service returned in the previous step passing for parameters the user constraints regarding the number of samples, the interval when the values were stored and their granularity.
- By doing so, the real values registered by sensors associated to the given services are retrieved, and the result set of each service's response is composed by pairs of real value and sampling date.
- Thank to these results, we are able to compute the service output penalty.

7.2 Service output penalty

Service output penalty depends on the quality of the output values provided by services. The way quality is assessed depends on the service interface: for services acting on actuators no penalty is applied since they don't provide any output value, whereas for services monitoring sensors quality is assessed according to the output type that, as said in the service ontology description in part 6.1.1 of this document, can be *Last One*, *Last X* or *Interval*.

7.2.1 Last One

Last One services simply return the most recent value collected by the associated sensors and its sampling date.
Intuitively, service quality depends on the sampling date: the older is such date, less accurate is the service. So, the simplest way to compute service penalty would be to apply a penalty directly proportional to the difference from the returned value's date and a reference one.
Our procedure at first finds the most recent date within the ones associated to all values returned by services. Then it uses that value to evaluate the quality of all services. In this way the service, whose returned value is the most recent within the ones provided by the result set, and that satisfies the user query, will get no penalty. The others will receive a penalty directly proportional to the difference between that date and the ones associated to their returned values.

For example, if we have two *Last One* services whose returned values are associated respectively to 10:32 PM and 2:15 AM of January 1, 2004, the former won't get any penalty, whereas the latter will be penalized by a value which is function of the time difference between 10:32 PM and 2:15 AM.
More in detail, this penalty function is computed by following these steps:

- We find the most recent date within the set of services' results, which will be the start date (see Algorithm 5 lines 1 to 6).
- For every service we calculate the time difference as the difference in seconds between the start date and the date registered for their output value (see Algorithm 5 lines 7 to 10).
- The maximum value of the time difference is saved and is used to calculate the rating of every service as 1-(time difference / max)*lastone_penalty, being lastone_penalty a constant value (see Algorithm 5 lines 11 to 14).

In this way the service providing the most recent value will get a rating of 1, whereas the rating of all the other ones will be lower than 1.
It is worth mentioning that we discarded to use the current date as reference date, as we firstly thought, because by doing so we would have heavily penalized services even if all of them had old sampling dates. Suppose sensors associated to all services satisfying a certain query stopped sampling data one year ago: by using the above mentioned method to compute service penalty, all of them would get a very high one and probably no service would be considered suited to answer the user query.

Algorithm 5 Last One algorithm

```
for all k = 1:total number of services do
    extract the corresponding date
    if recent date¿date extracted then
        update recent date
    end if
end for
for all k = 1: total number of services do
    extract the corresponding date
    time difference =calculate the difference between recent date and the corresponding date
end for
max=find maximum time difference;
for all k = 1: total number of services do
    rating(k)=1-(time difference/max)*penalty;
end for
```

7.2.2 Last X

Last X services, given in input the desired number of output values, return to the user the most recent values collected by the associated sensors and their sampling dates. If the number of requested output values is greater than the number of values collected by the service, SCoReS returns all the available values. Instead if the service has more values than the ones requested by the user, SCoReS returns the most recent ones.
There are two different parameters that we use in order to catalog and order SCoReS simple services:

- The number of values, or real time data, registered by the associated sensor.
- The date associated to the most recent value.

So we expect that a service able to retrieve exactly the same number of values as the one specified by the user should have higher quality than a service that satisfies the user request only partially by giving him fewer values than the ones he requested. That quality should depend on the number of returned values. We also expect that the penalty associated to a service that returns five results, when the user asked for ten, should be greater than the one associated to the same service if the user have asked for six results.
Moreover, we expect that, if two services return the same number of output values, their quality should vary according to their output values' sampling dates: a service returning values that are more recent than the values returned by another one have higher quality than the latter.
Such expectations are fulfilled by the service output penalty computation algorithm. In fact, for services having *Last X* output type, we follow these steps:

- For every service we count the number of items in the result set (see Algorithm 6 lines 1 to 3).
- We use the number of samples given by the user as a comparison value. In this way we can calculate the rating as the difference between the comparison value and the real number of items for each service (see Algorithm 6 lines 5 to 8).
- Then we consider the most recent date for every service. If a service has a result set with two or more items, we compare all the dates present in the pair real value - date, and we extract the most recent one (see Algorithm 6 line 9).
- The algorithm catalogs every service in a set of classes, where every class corresponds to the number of samples. For example a service with four

samples is inserted in the Class Four; instead the service with ten samples is in Class Ten.

- For every class we extract the most recent date.
- We finally compute an additional penalty for each service based on the difference between their most recent date and the most recent one in their class. This procedure is similar to the one followed for the *Last One* services, with the particular use of service's class. In this way we can find the best service for every range of items (see Algorithm 6 line 13).

Algorithm 6 Last X algorithm

```
1: for all k = 1: total number of services do
2:     total element(k)=count number of items in the result set for the service (k);
3: end for
4: for all k = 1: total number of services do
5:     rating(k)=1;
6:     if total element(k)¡sampleNumber then
7:         rating(k)=total element(k)/sampleNumber;
8:     end if
9:     lastvalue(x)=extract most recent date
10: end for
11: for all k = 1: max(total element) do
12:     for all services having total element=k do
13:         calculate with the same procedure of Last One the temporal penalty
14:     end for
15: end for
```

7.2.3 Interval

Interval services return values that are within an interval specified by the user in the input parameters. In order to assess services' quality, a third input parameter named granularity is required, which specifies within how many seconds at least one output value should be. For example, by setting the granularity parameter to 60 seconds, we expect at least one output value every 60 seconds. By doing so, we can subdivide the interval specified by the user in subintervals having the same duration as indicated by the granularity parameter, and thus we can count the number of values that fall in each subinterval and apply penalties to services having subintervals with zero values (gaps).

Intuitively, penalties should be directly proportional to the number of gaps for services that have the same interval and granularity. Anyway, simply counting the number of gaps won't be enough to precisely assess service quality: suppose to query two Interval services providing the same interval but different granularities for the input parameters, say 30 seconds for the former and 300 seconds for the latter, and finding out that both results have two gaps. It would be unfair to apply the same penalty to both services, since the former has many more subintervals than the latter, and so the result it provides is much more accurate than the one provided by the other service. Now, suppose to query two *Interval* services providing the same granularity but different intervals, say 10:00 AM to 2:00 PM of January 1, 2004 for the former and 8:00 AM to 4:00 PM of March 9, 2004 for the latter (both intervals' dates in the same day), and receiving results that have both three gaps: again, if we would apply the same penalty to both services we wouldn't consider that the former is less accurate since its interval is shorter than the one specified for the latter.

For these reasons, the best way to compute *Interval* service penalty is dividing the gap count by the number of subintervals, and using the obtained value, which is the percentage of gaps with respect to the interval, to assess service quality.

The steps in our analysis are the following ones:

- We divide the interval between the start and the end date in several time spans, according to the granularity chosen by the user (see Algorithm 7 line 1).
- For every service we count the number of items present in every time span of the result set, creating an aggregated result (see Algorithm 7 lines 2 to 11).
- We then analyze the aggregated result and calculate a penalty as the number of empty time spans divided by the total number of time spans (see Algorithm 7 lines 12 to 20).

Algorithm 7 Interval algorithm

```
1: number of time span= (end date - start date)/granularity
2: for all k=1: number of time span do
3:    for all for i=1: total number of services do
4:       for all for j:total number of items in the result set do
5:          extract corresponding date for the item(j)
6:          if corresponding date is inside the time span (k) then
7:             aggregator(current service, k)++
8:          end if
9:       end for
10:   end for
11: end for
12: for all k=1: total number of services do
13:    timeSpanAbsence=0;
14:    for all i=1: number of time span do
15:       if aggregator(k,i)==0 then
16:          timeSpanAbsence++
17:       end if
18:    end for
19:    rating(k)=1-timeSpanAbsence/ number of time span
20: end for
```

7.3 Location penalty

As previously said, location penalty depends on the distance between the location where the sensor associated to the service is and the one specified by the user. Intuitively, we expect that services whose associated sensors are distant from the specified location should receive a penalty greater than the one given to services whose sensors are close to that. For example, if a service monitors a room that is next to the one we specified, we expect a very low penalty, whereas we expect a greater penalty for a service associated to a room which is in another building.

It is also natural to consider that the broader the location monitored by a service, the greater the penalty. For example, given by the user a certain room in the service list query, a service that monitors the room's building should have greater penalty than another one that monitors the floor where such room is.

It is also worth noting that, if a broader location is specified by the user, the location penalty should be relaxed, and services associated to locations that are more specific than the specified one shouldn't get any penalty. For example, a service associated to a floor different from the one specified by the user should have lower penalty than the one it would have got if the user had specified a room in the service list query instead (and the room was in a floor different from the one associated to the service), and a service monitoring a room which is on the specified floor should have no penalty.

That said, we are able to deal with such expectations by exploiting the location ontology: since each location can be a room, a floor, a building or a campus, and relationships between locations exist in the ontology, we can represent the whole ontology as an unweighted and undirected graph having for nodes the concepts' instances and for edges their relationships (see Figure 8). By doing so, we can compute the service location penalty by finding the shortest path between the location associated to a service and the location specified by the user, then counting the number of edges in the path found that connect locations that are less specific or as specific as the specified one (e.g. a floor is less specific than a room, as specific as another floor, and more specific than a building), and finally multiplying a constant value (which is in the range [0-1]) by itself as many times as the value computed in the previous step (see Algorithm 8).

Algorithm 8 Location algorithm

```
1: Edge count=0
2: for all edge: edges composing shortest path do
3:    if !lessSpecific(edge.fromNode, user location) and! lessSpecific(edge.toNode, user location) then
4:       Edge count++
5:    end if
6: end for
7: Location penalty = LOCATION PENALTY CONSTANT$^{Edge\ count}$
```

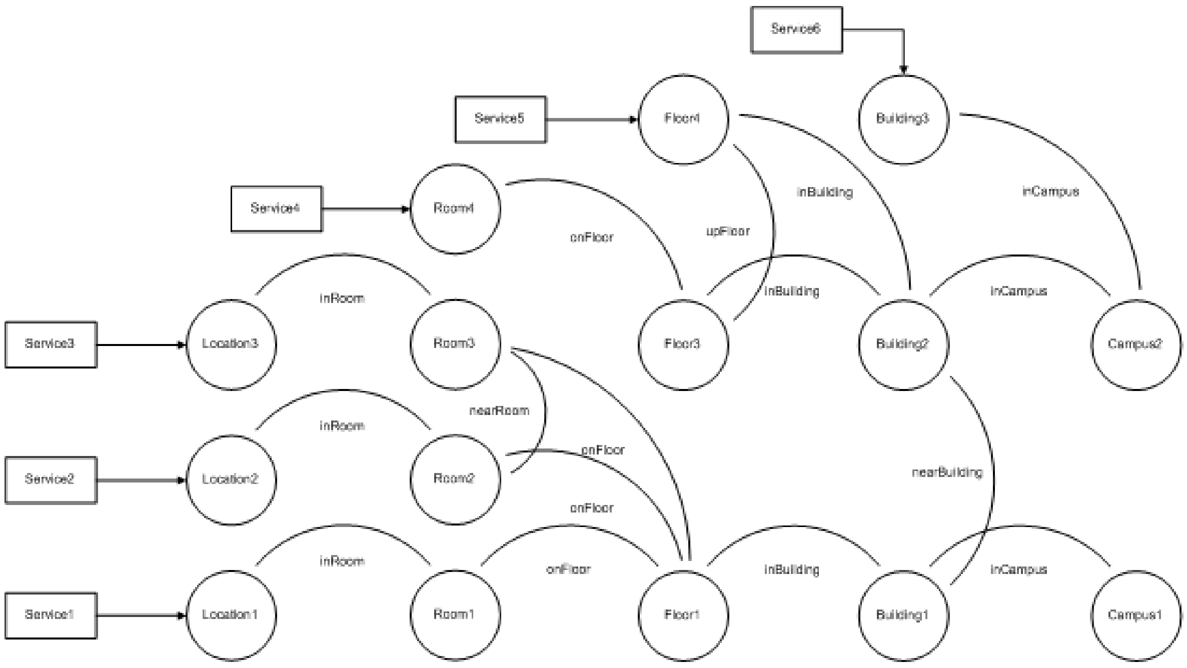


Figure 8: Example A for location penalty

For example, if we asked for services belonging to Room3, Service3 won't get any location penalty since its associated location is exactly Room3, whereas Service2 will get a location penalty that is equal to LOCATION PENALTY CONSTANT[1] since in the shortest path between Room3 and Room2 (which is the location where Service2 is) there is one edge and Room2 is as specific as Room3 (see Figure 9).

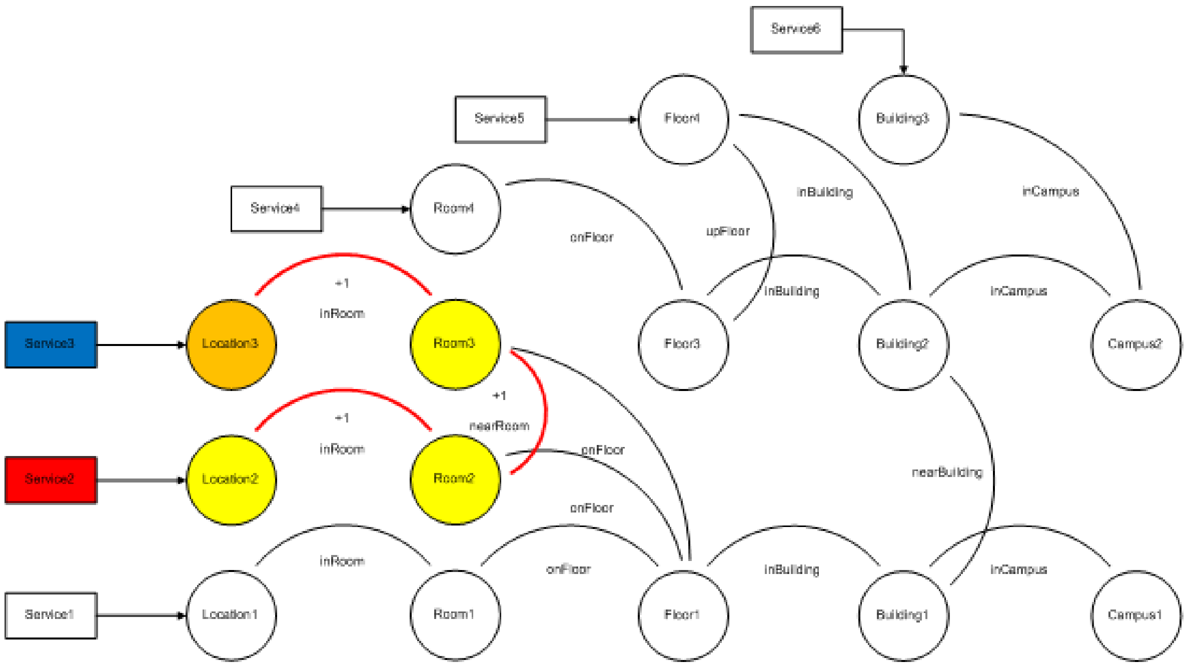


Figure 9: Example B for location penalty

If we asked for services belonging to Floor3 instead, Service4 won't get any penalty because the shortest path is made only of an edge that connects Room4, which is more specific than Floor3, Service5 will get a penalty equal to LOCATION PENALTY CONSTANT^1 since the only edge of the shortest path links Floor4 that is as specific as Floor3, and Service1, Service2 and Service3's penalty will be equal to LOCATION PENALTY CONSTANT^3 because there are only three edges in their shortest path that connect locations less specific or as specific as Floor3, which are Floor3-Building2, Building2-Building1 and Building1-Floor3 (see Figure 10).

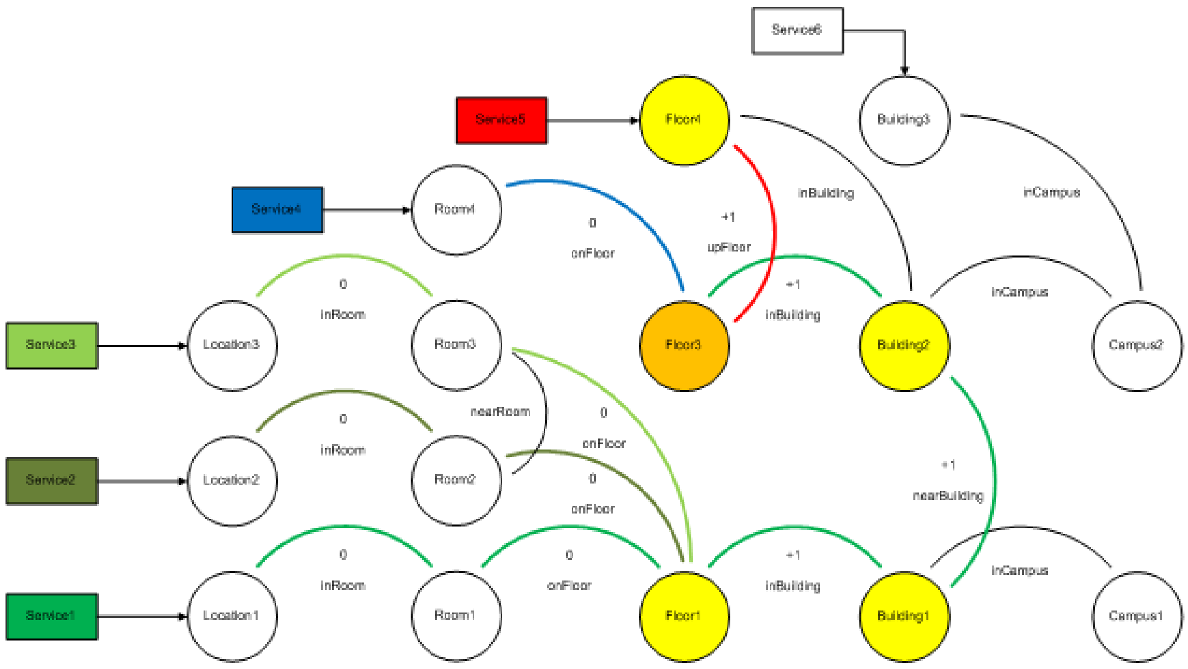


Figure 10: Example C for location penalty

7.4 Time output penalty

Time output penalty depends on the degree of compatibility between services having different time output property values. For example, if we asked for services having time output value equal to *Last X*, we could also be interested in services having Interval as time output property value, because by setting the properties of an Interval service opportunely we can get a similar result; on the contrary, we won't be interested in *Last One* services because they provide only the most recent value, and so they are unsuited in answering a query asking for the last n values.

That said, we can summarize the services' time output property similarity with Table 2.

	Service time output property		
Query time output property	**Last One**	**Last X**	**Interval**
Last One	Exact	Close	Close
Last X	Mismatch	Exact	Close
Interval	Mismatch	Close	Exact

Table 2: Time output similarity

The algorithm simply compares each service time output property with the one specified in the query parameters by using that table, and assigns a penalty equal to 1.0 when the correspondence is exact, to the time output penalty constant value when it is close, and to 0.0 when there is no correspondence, thus discarding that service.

7.5 Algorithm tuning

By defining location and service output penalties in the previous sections, we introduced 3 constants: service output penalty, location penalty and time output penalty. In order to get the best results with SCoReS similarity algorithm, an accurate tuning of these constants is required.

The first step consists in defining three queries asking for services having the requirements mentioned in Table 3, and collecting all services satisfying these constraints:

Name	**Time output**	**Neighbors extension**	**Location**	**Specific parameters**
Query 1	Interval	No	-	Range: from 28/2/2004 10:30 AM to 28/2/2004 6:30 PM Granularity: 1 hour
Query 2	LastX	No	Floor22	Samples: 14
Query 3	Now	Yes	Room27	-

Table 3: Example A for testing

Notes: for queries asking for neighbors extension, we simply ignore the specified location in the query constraints, whereas for queries without that constraint we include services whose associated location is more specific than the one specified in the query constraints. For queries asking for exact time output, we include only services having the time output property value identical to the one specified in the query constraints, whereas for queries without that constraint we simply do not check the services' time output property. Due to lack of data collected by all services other than the ones having Light kind and Electricity category in our test-bed, we introduce in the requirements of all queries the following further constraints: Kind: Light, Category: Electricity.

After that, for each query we order the returned services by hand according to how meaningful the values provided by these ones are (the first position is given to the services whose returned data are the most meaningful ones), and then we mark as discarded the ones whose values are not meaningful enough. These results, named training set, will be used to tune the algorithm.

Having done so, we execute SCoReS similarity algorithm by providing in input the same constraints and compare its output results with the training set. In order to do such comparison we plotted a precision-recall graph following these criteria:

- We calculate the resulting rating for every service, with a value between 1 and 0.
- We compute a valid matrix, adding a 0 if the rating is lower that a certain threshold, 1 otherwise.
- Running an external function [31], it is possible to draw a precision-recall graph and a ROC curve.

Finally, we iterate the previously mentioned step varying the algorithm's constant values until we get the best precision-recall ratio, which is shown in Chart 11:

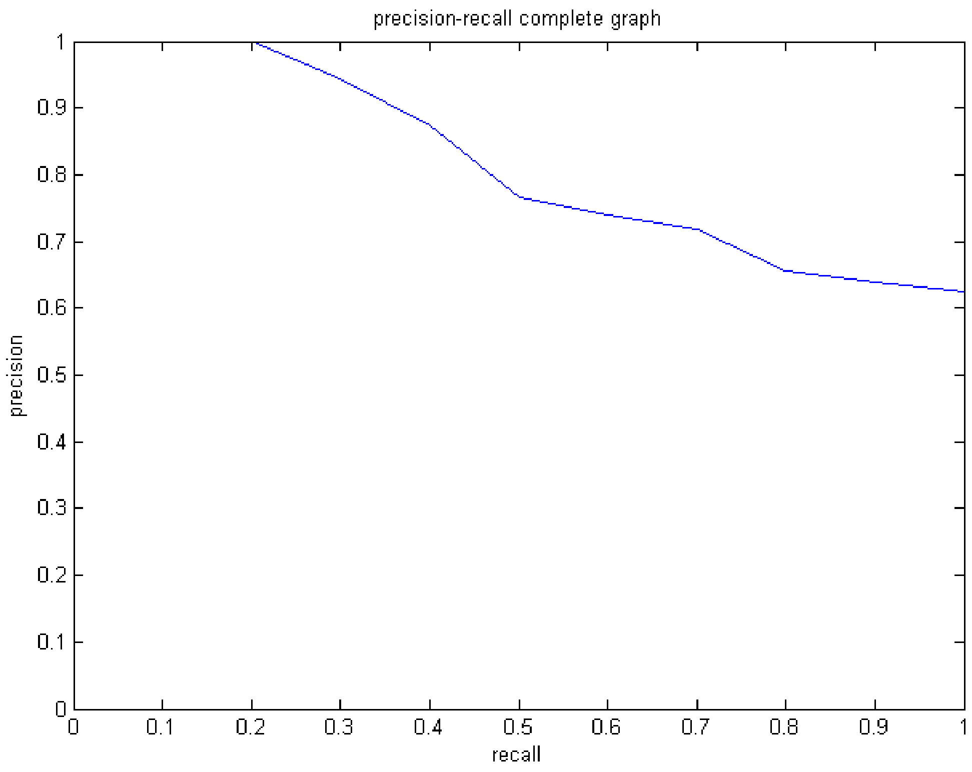


Figure 11: Precision-Recall Graph for tuning SCoReS similarity algorithm

In order to be sure that SCoReS similarity algorithm works as expected, we have also defined three more queries similar to the previous ones whose constraints are shown in Table 4.

Name	**Time output**	**Neighbors extension**	**Location**	**Specific parameters**
Query 4	Interval	No	-	Range: from 28/2/2004 6:30 PM to 29/2/2004 2:30 AM Granularity: 1 hour
Query 5	LastX	No	Floor12	Samples: 11
Query 6	Now	Yes	Room111	-

Table 4: Example B for testing

We then order and classify the results obtained by these queries the same way we did for the previous ones. The result of this procedure, named test set, will be used to test the algorithm's goodness.

In fact, this time we execute SCoReS similarity algorithm with the test set's query constraints only once keeping the constant values fixed, and then we compare the algorithm's results with the test set, obtaining Chart 12.

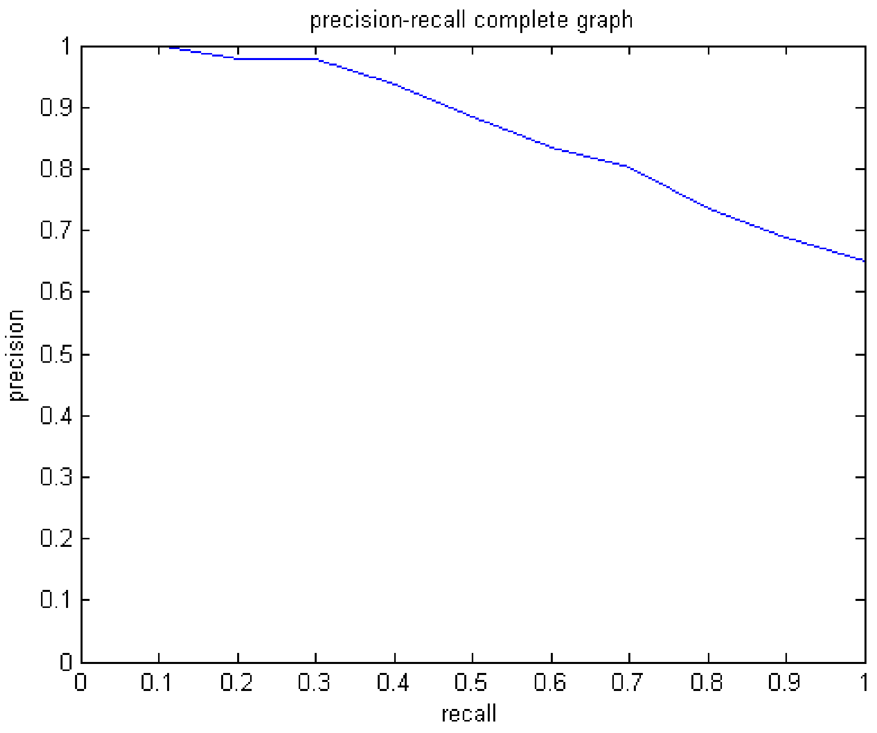


Figure 12: Precision-Recall Graph for the Test Set

All these results are obtained by using a threshold of 0.5. The threshold is the minimum ranking value that a service must get in order to be listed in the final results set.

We have chosen this specific value after controlling the performance of SCoReS similarity algorithm with different threshold (see Charts 13, 14 and 15). With a range between 0.3 and 0.9, we have analyzed the precision-recall graph for the three types of services:

- For *Last One* and *Last X* we have found that a value from 0.7 to 0.5 shows good results.
- For *Interval* we need to lower the threshold to a value between 0.5 and 0.4.

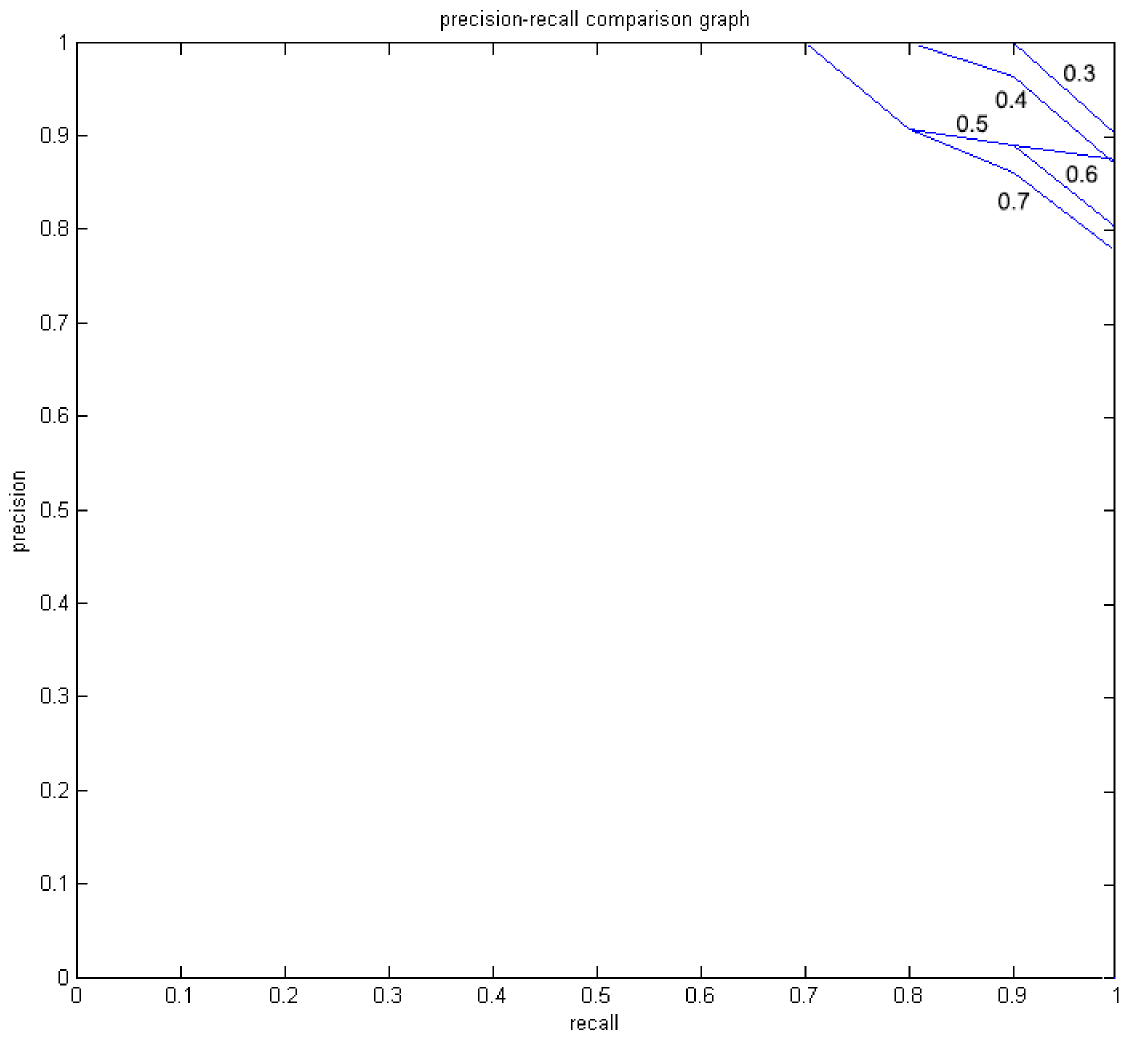


Figure 13: Precision-Recall Graph for Last One services, with different thresholds

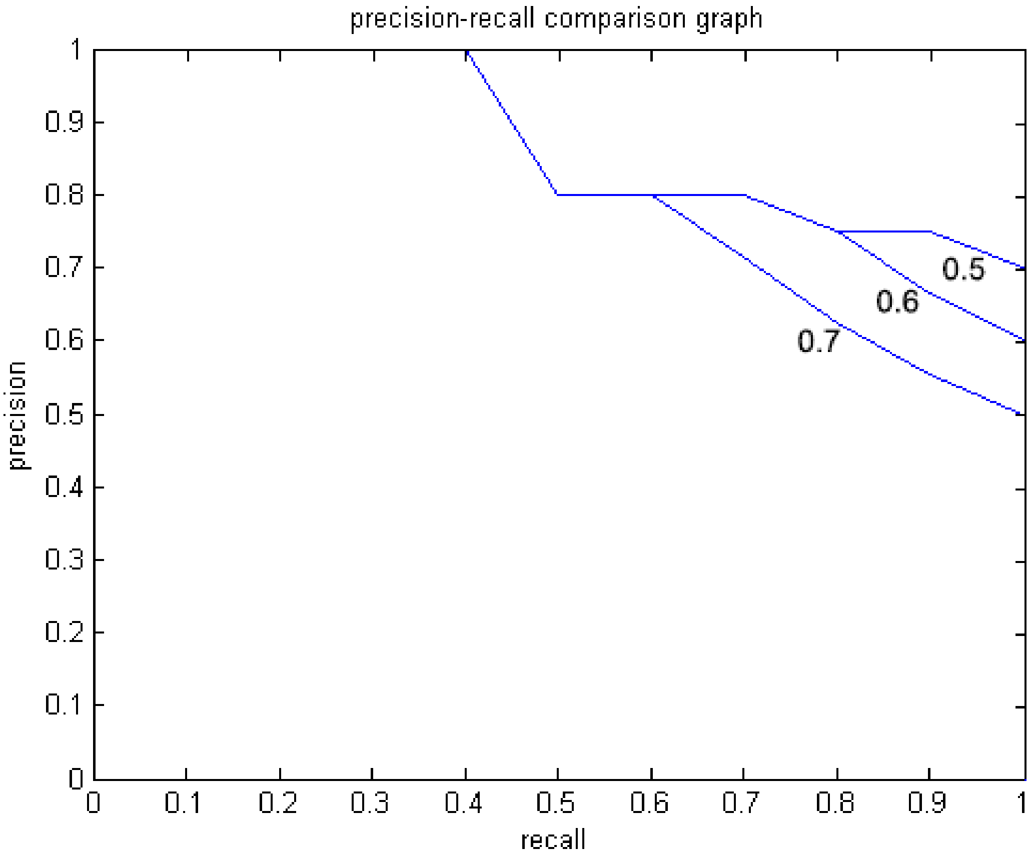


Figure 14: Precision-Recall Graph for Last X services, with different thresholds

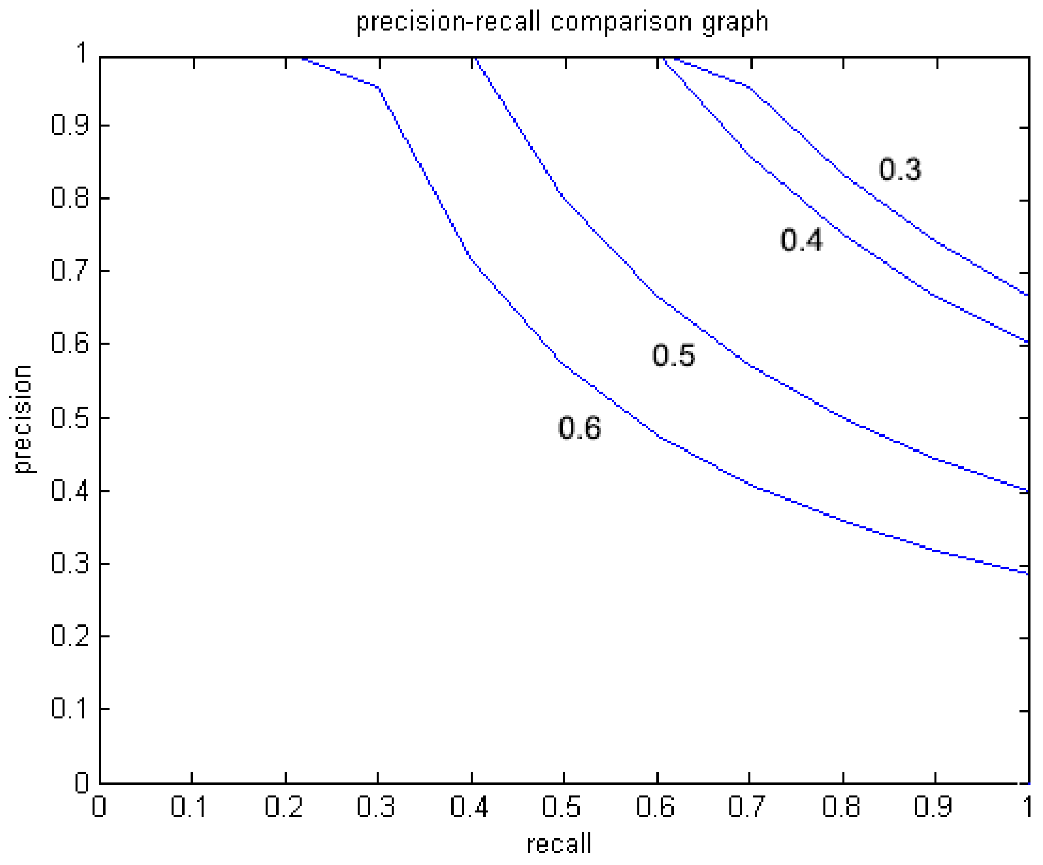


Figure 15: Precision-Recall Graph for Interval services, with different thresholds

For values of threshold under 0.3 we have registered poor quality results. In fact SCoReS similarity algorithm includes a lot of services that do not satisfy the query constraints at all. Vice versa for thresholds above 0.7, the result set is very small or empty.

The core feature of SCoReS similarity algorithm is to create a valid list for the service retriever component. Too many results would confuse the user, who could difficultly find the service suited for his needs. Few results would limit the variety in the response, forcing an unexpert user to create a lot of similar queries in order to find the right parameters for his needs.

In order to have the same threshold for the three different types of services, we chose to implement SCoReS with a standard value of 0.5. With this number we can offer a not too wide result set, with different services that can be right for the sent query.

8 Service composition

As said in the previous chapters, in SCoReS we have simple services, which provide data associated to a specific sensor, and composite services, which are services that provide complex functionalities, like getting values from certain sensors and using them to act on some actuators.

In order to provide these functionalities, the design of composite services implies:

- Getting and setting data by invoking other services, which could be simple services or composite services as well.
- Manipulating data by combining them and/or performing simple operations, like computing the average value or the maximum.

Moreover, the task of building composite services should be done by local managers, who usually don't have much technical expertise. For this reason the user should be allowed to create new services in an intuitive way, involving as less coding as possible.

The approach that best meets these requirements is the creation and execution of a workflow: workflows are easy to understand by inexperienced people, since they can be designed by using tools that graphically show the control flow, but nevertheless are very powerful since they allow to express a great variety of constructs. For these reasons, by using workflows during the composite services creation process, this procedure can be made extremely intuitive.

After considering many preexisting workflow languages by examining [26], [18], [25] and [21], we chose not to use any of them and to build our own workflow language. We made this choice because adapting a preexistent workflow engine capable to run one of these languages to our purpose would have required a greater effort than building an ad-hoc workflow language and its engine from scratch.

SCoReS workflow language implements the most important workflow patterns available in other languages, like WS-BPEL, as stated in Table 5:

Pattern	**Description**	**Implemented?**
Sequence	Execute two or more activities in sequence	Natively
Parallel Split	Execute two or more activities in any order or in parallel	No
Synchronize	Synchronize two or more activities that may execute in any order or in parallel; do not proceed with the execution of the following activities until all these preceding activities have completed	No
Exclusive Choice	Choose one execution path from many alternatives based on data that is available when the execution of the process reaches the exclusive choice	Natively
Simple Merge	Wait for one among a set of activities to complete before proceeding; it is assumed that only one of these activities will be executed	Natively
Terminate	Terminate execution of activities upon defined event or status change	No
Multiple Choice	Choose several execution paths from many alternatives	No
Conditional Choice	Choose one execution path from many alternatives according to discriminated status conditions	Indirectly
Synchronizing Merge	Merge many execution paths; synchronize if many paths are taken; do the same as for a simple merge if only one execution path is taken	No
Multiple Merge	Wait for one among a set of activities to complete before proceeding; if several of the activities being waited for are executed, the simple merge fires each time that one of them completes	No

Table 5 – continued from previous page

Pattern	Description	Implemented?
Discriminator	Wait for one among a set of activities to complete before proceeding; if several of the activities being waited for are executed, the discriminator only fires once	Indirectly
N-out-of-M Join	Same as the discriminator but it is now possible to wait until more than one of the preceding activities completes before proceeding by setting a parameter N to some natural number greater than one	No
Arbitrary Cycle	Do not impose any structural restrictions on the types of loops that can exist in the process model	Natively
Implicitly Terminate	Terminate an instance of the process if there is nothing else to be done	Indirectly
Multiple Instances without synchronizing	Generate many instances of one activity without synchronizing them afterwards	No
MI with a prior known design time knowledge	Generate many instances of one activity when the number of instances is known at the design time (with synchronization)	Indirectly
MI with a prior known runtime knowledge	Generate many instances of one activity when a number of instances can be determined at some point during the runtime	Indirectly
MI without a prior runtime knowledge	Generate many instances of one activity when a number of instances cannot be determined	Indirectly

Table 5 – continued from previous page

Pattern	Description	Implemented?
Deferred Choice	Execute one of a number of alternative threads. The choice which thread is to be executed is not based on data that is available at the moment when the execution has reached the deferred choice, but is rather determined by an event	No
Interleaved Parallel Routing	Execute a number of activities in any order (e.g. based on availability of resources), but do not execute any of these activities at the same time/simultaneously	No
Milestone	Allow a certain activity at any time before the milestone is reached, after which the activity can no longer be executed	Indirectly
Cancel Activity	Stop the execution of an enabled activity	No
Cancel Case	Stop the execution of a running process	Indirectly
Cancel Wait	Continue execution of a running process without prior completion event	No

Table 5: List of patterns supported in SCoReS workflow engine

Note: with the term "natively" we indicate that SCoReS workflow language has a construct which has the same behavior of that pattern, whereas with the term "indirectly" we indicate that the behavior of that pattern can be simulated by opportunely combining one or more of SCoReS constructs.

We chose not to implement all of them because at this stage we just want to provide a prototype workflow engine capable of satisfying the most common needs. In future we will expand SCoReS workflow engine capabilities by implementing all patterns.

As for the workflow activities, we decided to make available two particular types:

- Service invocation activities, which are used to invoke services and store their result, if any, into a workflow variable.

- User code execution activities, which can be used to execute arbitrary code written by the end user in order to perform operations on data.

Taking inspiration from WS-BPEL, we distinguish workflows into executable workflows and templates:

- Executable workflows are the ones whose invoke statements contain references to actual services and thus, as the name suggests, can be directly executed as composite services.

- Templates, on the other hand, are workflows whose invoke statements contain references to queries that, if executed, provide a list of services matching the specified requirements. They can't be executed. Their purpose is to allow the end user to choose from the result set of each query stored in the template a service and, after having done so, create an executable workflow that has the same structure as the template, but has the previously chosen services instead of the service queries. In this way it is possible to create executable workflows having the same control flow structure but different invoked services (hence the name template).

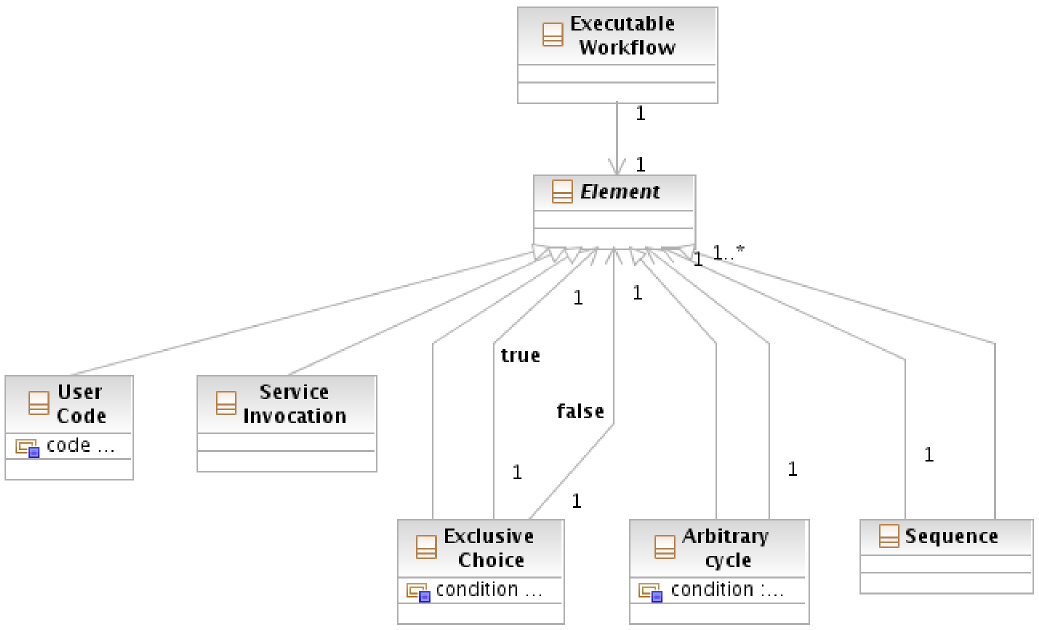


Figure 16: Elements in an executable workflow

As shown in Figure 16, an executable workflow contains an element, which can be a Sequence, an Arbitrary cycle, an Exclusive choice, a Service invocation or User code. Sequence, Arbitrary cycle and Exclusive choice elements contain one or more elements that can have any of the above specified categories. This way it is possible to nest a construct inside another until either a Service invocation or a User code one is specified.

By doing so, it is possible to represent a workflow as a tree having:

- The top element as root.
- Sequence, Arbitrary cycle and Exclusive choice elements as nodes.
- Service invocation and User code as leaves.

This way workflow execution essentially consists in a depth first analysis of the tree.

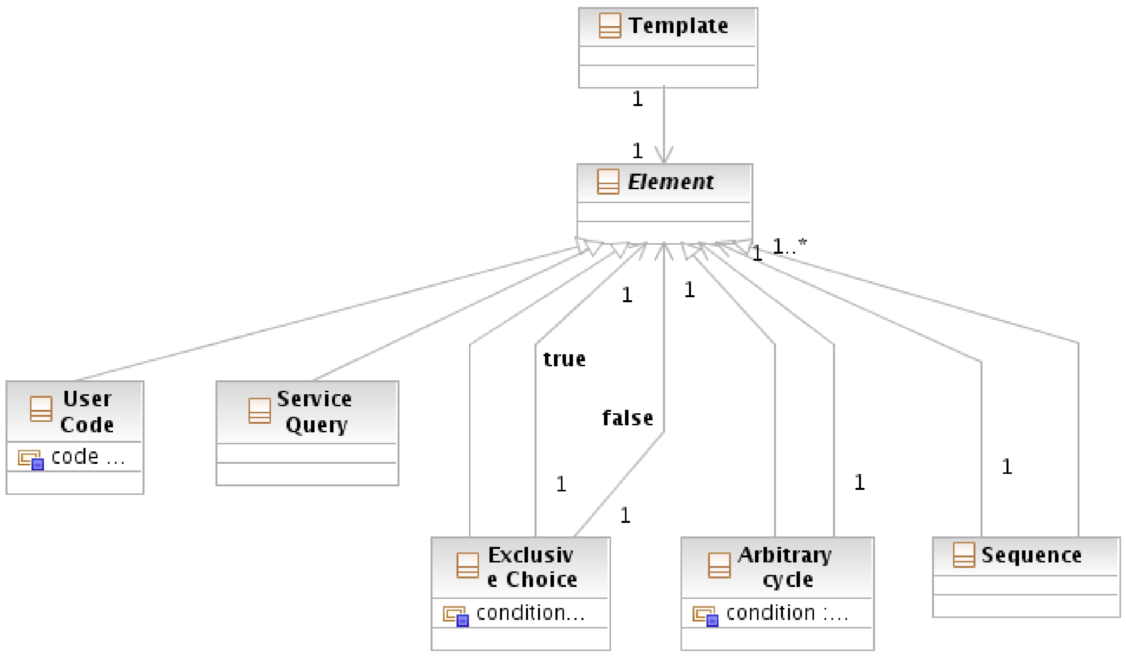
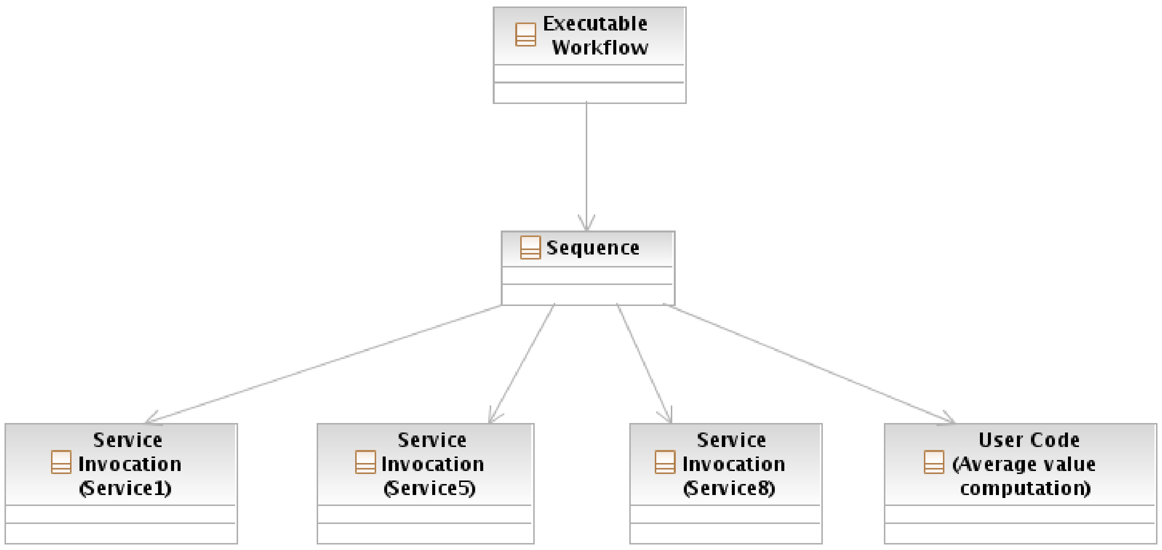


Figure 17: Elements in an Template workflow

Templates are almost identical to executable workflows: the only difference is that, instead of Service invocation elements, there are Service query ones, which contain a query that will be executed whenever the user choose to create an executable workflow based on that template (see Figure 17. Templates will be transformed into executable workflows by replacing Service query blocks with Service invocation ones invoking services selected by the user from the query results.

To better understand how SCoReS workflow language works, suppose that we need to know the temperature of a room without sensors. We can easily solve that problem by creating a service that invokes simple services monitoring the temperature of rooms adjacent to the required one (say Service1, Service5 and Service8) and then returns the average value of the ones provided by the invoked services. This task can be easily accomplished by creating a composite service which uses the workflow shown in Figure 18.


Executable Workflow
Sequence
Service Invocation (Service1)
Service Invocation (Service5)
Service Invocation (Service8)
User Code (Average value computation)
Figure 18: Example A of executable workflow

As we can see, the workflow uses a Sequence element that contains three Service invocation elements for invoking the services, and an User code element in order to compute the average value.

(**Note:** elements contained into the Sequence one are evaluated from left to right)

If we knew in advance that Service1 generally provides temperature values very similar to the ones in the required room, we could modify the procedure by firstly invoking Service1, then checking their returned value and, if it is either missing or wrong (i.e. a value too low or too high due to a malfunction of the associated sensor), by computing the average value between the values returned by Service5 and Service8. In this case the associated workflow would be modified in this way (see Figure 19):

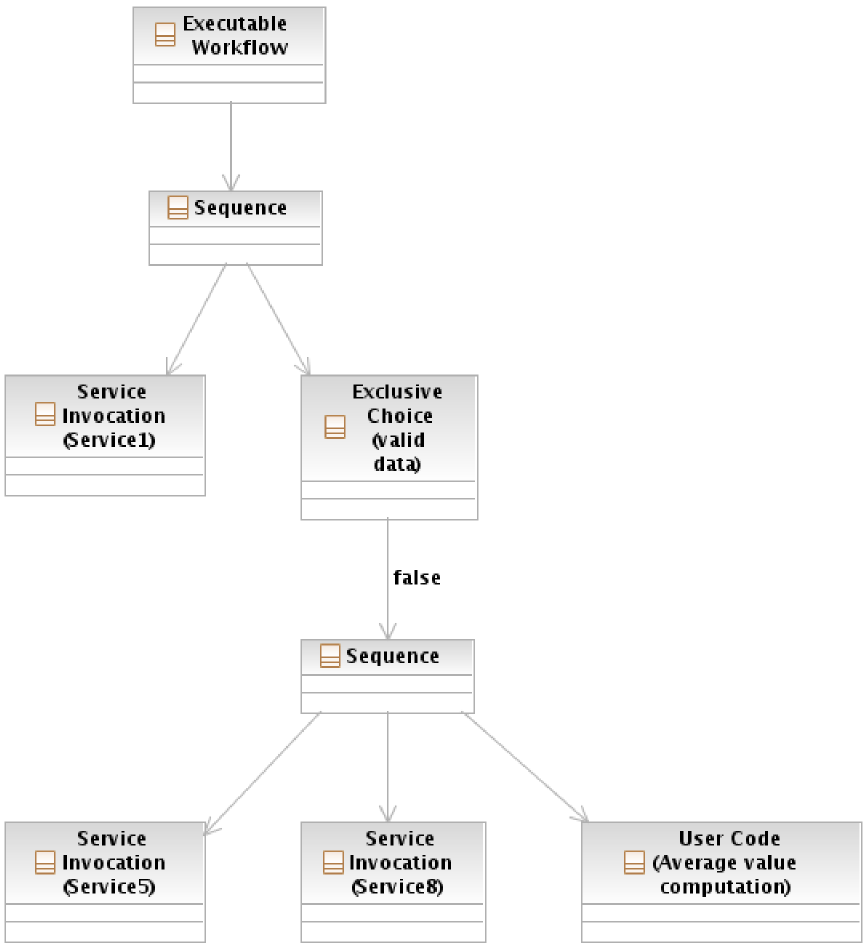

Executable Workflow
Sequence
Service Invocation (Service1)
Exclusive Choice (valid data)
false
Sequence
Service Invocation (Service5)
Service Invocation (Service8)
User Code (Average value computation)

Figure 19: Example B of executable workflow

This time the workflow uses a Sequence element that contains a Service invocation element for invoking Service1, and an Exclusive choice to evaluate if data returned by Service1 are valid. The false branch of the Exclusive choice element, which is executed whenever the choice condition is false, is associated to another Sequence element that contains two Service invocation elements for invoking Service5 and Service8, and an User code element that computes the average value of the results provided by the previously mentioned services.

(**Note:** elements contained into the Sequence ones are evaluated from left to right)

9 Implementation of SCoReS

In order to have a scalable and distributed infrastructure, we chose to design and implement our project using the Service Oriented Architecture paradigm. Figure 20 shows the main components of our architecture.

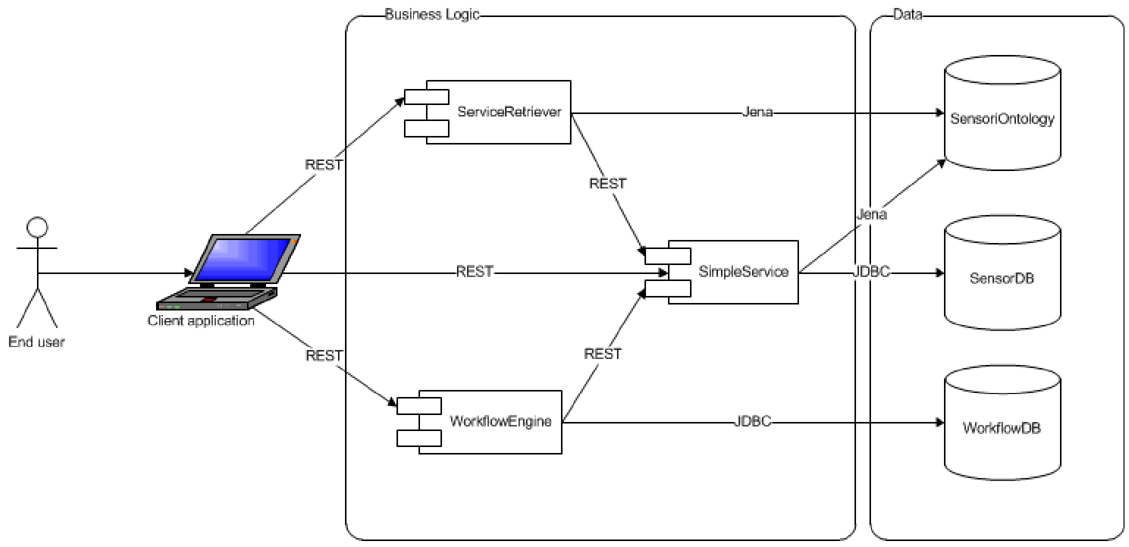


Figure 20: General structure of SCoReS

As stated in the diagram, backend functionalities are provided by the following web services:

- **SimpleServices**, which provides data belonging to a certain service given the service id.
- **ServiceRetriever**, which is responsible for finding services that satisfy the specified conditions and ranking them according to their similarity.
- **WorkflowEngine**, which is responsible for saving, retrieving and executing composite services.

More in detail, SimpleServices interacts with an ontology, which contains information about services, sensors, locations and their relationships (**SensoriOntology**), and a database where data collected from sensors are stored (**SensorDB**).
WorkflowEngine, instead, interacts with a database containing workflows associated to composite services (**WorkflowDB**).

Finally, ServiceRetriever invokes SimpleServices and interacts with SensoriOntology.
All these services use a shared library, which is used to interact with the ontology, build and parse JSON objects, invoke external services and share the workflow constructs and data models.

9.1 Service Retriever Module

ServiceRetriever web service is composed of the following classes, as shown in figure 21:

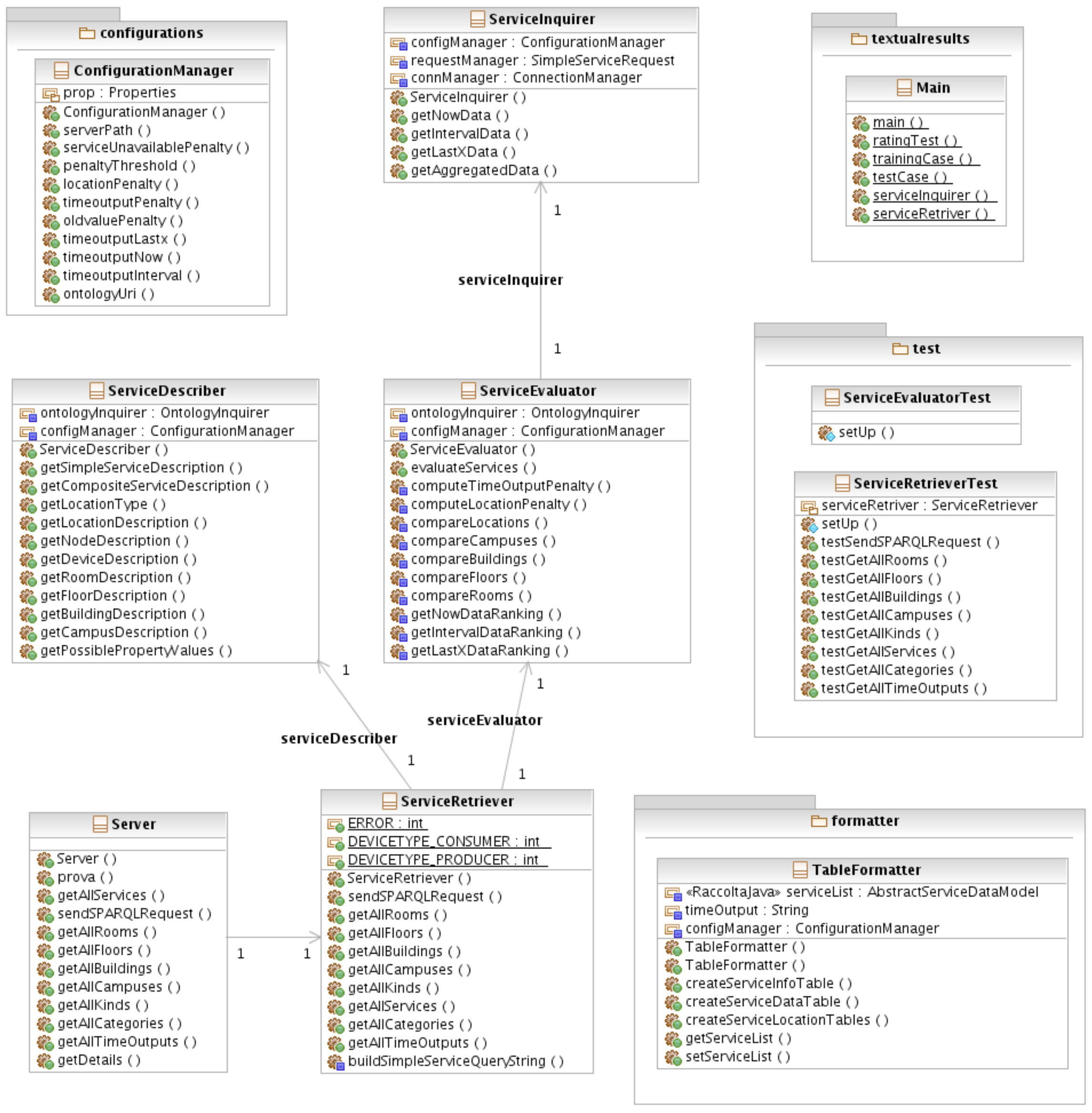


Figure 21: General structure of Service Retriever module

Inside the module there is a ConfigurationManager class, used to read the configuration file and get values associated to some parameters. Every single parameter configure a part of our core functionalities, that the developer can personalize for its needs.

9.1.1 Server class

The Server Class manages the entire REST requests sent by the user. Every request has the same base URL, but appending at the end different paths can generate different methods:

- In order to search specific services is possible to utilize two different requests, one parametric, and the other, raw, for more advanced users.
- To find the right value for every parameter there are several support requests that describe all the possible correct values.
- For the details of a specific concept present in the ontology it is possible to request a JSON description.

All these methods generate an error if the request is not well formatted or if a mandatory parameter is missing. For the complete list of the entire set of operations see section B.

The parametric search functionality is very detailed and the user can find specific services by tuning the right parameters.

There are two types of support requests:

- A general one, which describes all the possible values for a parameter.
- A specific one, which searches the values that correspond to some characteristics chosen by the user.

9.1.2 Service Retriever class

This class is a general interface that lets the client interact with the server by invoking useful operations.
Its methods can be divided in three different types:

- The query ones, which let the user request a list of services with particular characteristics.
- The parametric ones, that allow the user to receive a list of parameters that can be used as input parameters for the previous type of methods.
- The advanced ones, for expert users.

Each method returns a JSON string that can be formatted in two possible ways: as a list of objects, in which every item has properties and assertions extracted from the ontology, or an error string, containing details about the cause of such error.

For every object in the ontology there is a set of properties that are specific for it.

In section A, a complete list of general templates for every ontology concept is published, together with a list of all possible errors.

In order to retrieve a list of specific services, the user can invoke the method **getAllServices** with the following parameters:

- Kind, with a value within the response of the **getAllKinds** method.
- Category, with a value within the response of the **getAllCategories** method.
- NodeType, that must be either the Actuator or Sensor constant value.
- TimeOutput, with a value within the response of the **getAllTimeOutputs** method.
- ExactTimeOutput, either true if the user wants only services having time output property equal to the previously defined one, or false if also services having time output property different but compatible with the previously defined one should be considered (this parameter is considered only if the previous one is specified).
- location, with a value within the response of the **getAllBuildings**, **getAllCampuses**, **getAllFloors** or **getAllRooms** methods.
- extendToNeighbor, either true if the user also wants services near the specified location, or false if he doesn't (this parameter is considered only if the previous one is specified).
- sampleNumber, used to indicate the minimum number of results that a service with LastX timeOutput property should return (if not specified it is set to 1).
- from, used to indicate the starting date that a service with Interval timeOutput property should consider in order to provide the desired result (if not specified the interval is lowerly unbounded).
- to, used to indicate the ending date that a service with Interval timeOutput property should consider in order to provide the desired result (if not specified the interval is upperly unbounded).

- granularity, used to indicate the duration (in seconds) of each sub-interval that a service with Interval timeOutput property should consider in order to provide the desired result (if not specified coincides to the interval).

Expert users can use **sendSPARQLRequest** method to send a preformatted SPARQL query and the desired result type. In this way it is possible to query the ontology in order to extract various types of data, which can also be obtained with the general methods described above.

9.1.3 Service Describer class

This class invokes *OntologyInquirer* and *ServiceEvaluator* methods in order to collect information about available services and possible query parameters, and return that in a structured way.

More in detail, the class constructor instantiates the *OntologyInquirer* class, which will be available for the whole class lifecycle.
The method **getPossiblePropertyValues** builds a SPARQL query that asks to retrieve all possible values of the *category*, *kind* or *timeOutput* data property according to the given parameter's value, invokes the *executeQuery* method of the *OntologyInquirer* instance and returns the results of this one.

The method **getLocationType** specifies the type of the given location (room, floor, building, campus or specific location) by simply invoking the *getLocationType* method of the *OntologyInquirer* instance.

There are also some methods for getting specific information about certain services, like the node or location descriptions. All these methods are implemented in the following way:

- At first the *executeQuery* method of the *OntologyInquirer* instance is invoked, having in input the SPARQL query specified by the user, in order to have a list of ids.
- Then specific methods of the *OntologyInquirer* instance for each result type are invoked for each id.
- Finally the returned values are put in a support class instance, this one is added in a collection and, when all ids have been processed, the collection is returned.

More in detail, these methods are the following:

- **getLocationDescription**, which uses *LocationDataModel* support class and invokes *getLocationData* for each service id multiple times passing respectively room, floor, building and campus for the location type parameter.
- **getNodeDescription**, which uses *NodeDataModel* support class and invokes *getCategory*, *getKind*, *getNodeType*, *getLocationId* and *getDeviceId* methods for each node id.
- **getDeviceDescription**, which uses *DeviceDataModel* support class and invokes *getDeviceType* and either *getConsumptionData* or *getProductionData* for each device id.
- **getRoomDescription**, which uses *RoomDataModel* support class and invokes *getNearRooms* and *getFloorId* for each room id.
- **getFloorDescription**, which uses *FloorDataModel* support class and invokes *getUpFloor* and *getBuildingId* for each floor id.
- **getBuildingDescription**, which uses *BuildingDataModel* support class and invokes *getCampusId* and *getNearBuildings* for each building id.
- **getCampusDescription**, which uses *CampusDataModel* support class and saves each campus id in a support class instance.
- Finally, **getServiceDescription**, which uses *ServiceDataModel* support class and invokes *getTimeOutput* and *getNodeId* methods for each service id.

9.1.4 Service Evaluator class

This class evaluates the quality of services' responses, provides for each service a score and filters out irrelevant results by implementing the service similarity algorithm described in part 7.

More in detail, the class constructor instantiates the *ServiceInquirer* and *ServiceEvaluator* classes that will be available for the whole class lifecycle.

The method **evaluateServices** starts with a list of unordered services and follows a sequence of steps in order to rate them from the best result to the worst:

- At first the **getNowDataRanking**, **getLastXDataRanking** or **getIntervalDataRanking** private methods are invoked according to the time output property value requested by the user. These three methods compute the service output penalty for all the provided services.

- Then for each service, if the user asked to relax the time output constraint, the **computeTimeOutputPenalty** private method is invoked and the time output penalty is applied.
- If the user asked neighbor extension, the **computeLocationPenalty** private method is invoked and the location penalty is applied too.
- After that, services are filtered according to their penalty and services whose penalty is below a certain threshold are removed from the service list.
- Finally the ranked and filtered service list is returned to the user.

In details, the penalty based on service data is computed as it follows:

- For the method **getNowDataRanking**, the most recent data receives the highest ranking.
- For the method **getLastXDataRanking**, the ranking is calculated by combining two aspects: the number of data returned by a service and their respective timestamp. The service with the highest number of items and the most recent data will receive the highest ranking.
- For the method **getIntervalDataRanking**, the ranking is calculated by aggregating the data for a given service in the time span decided by the user and calculating the intervals of time covered by such values. The service with the highest number of intervals covered will receive the highest ranking.

9.1.5 Service Inquirer class

This class uses the REST API in order to get data from simple sensor network services.

More in detail, methods **getNowData**, **getLastXData** and **getIntervalData** follow this procedure:

- Firstly invoking the **createRequest** private method with parameters given by the user in order to build the service URI.
- Then calling up the **getServiceResponse** private method that returns the service response in JSON format.
- Finally converting the response from JSON to, respectively, *NowDataModel*, *LastXDataModel* or *IntervalDataModel*, and returning that response to the user, or throwing either a *ServiceMalfunctionException* or a *ServiceUnreachableException* if problems arise.

9.2 Workflow Engine Module

WorkflowEngine web service is composed of the following classes, as shown in figure 22:

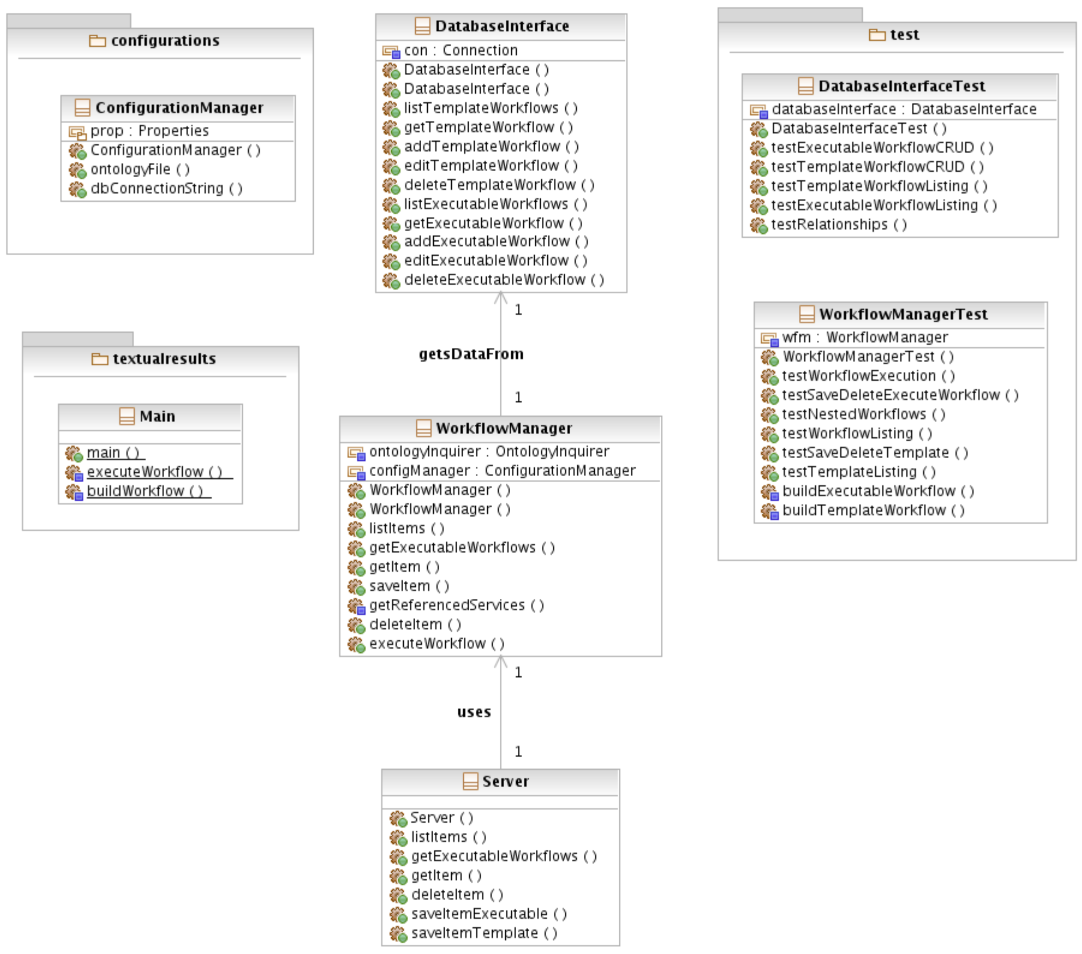


Figure 22: General structure of Workflow Engine Module

Inside the module there is a ConfigurationManager class, used to read the configuration file and get values associated to some parameters. Every single parameter configures a part of our core functionalities, that the developer can personalize for its needs.

9.2.1 Server Class

The main purpose of this class is providing a REST web service interface to this component. Thus, all of its functionalities are exploited by invoking *Workflow-*

Manager class methods.

9.2.2 Workflow Manager Class

This class uses *DatabaseInterface* and *ConfigurationManager* classes to respectively interact with the workflow database and read the service configuration file, and the *OntologyInquirer* class of the shared library to interact with the ontology.

More in detail, the class constructor instantiates the *DatabaseInterface*, *ConfigurationManager*, and *OntologyInquirer* classes. In particular, the latter is initialized by providing the path to the OWL file, that is provided by invoking the *ontologyFile* method of the *ConfigurationManager* class.

A second version of the constructor, which accepts a boolean as input, is used to let the application be run as a Java application, which is mandatory for being able to run JUnit tests:

- If the given value is true, the *DatabaseInterface* class is initialized by passing a connection string read from the configuration file.
- If it is false, its behaviour is exactly the same as the constructor with no parameters.

The **listItems** method checks the provided Integer parameter to determine if the user is asking for an executable workflow or a template, and invokes either the method *listExecutableWorkflows* or the *listTemplateWorkflows* one of the *DatabaseInterface* class, returning its result.

The **getExecutableWorkflows** method simply invokes the method *listExecutableWorkflows* of the *DatabaseInterface* class.

The **getItem** method checks the provided Integer parameter to determine if the user is asking for an executable workflow or a template, invokes either the method *getExecutableWorkflow* or the *getTemplateWorkflows* one of the *DatabaseInterface* class, converts the value given by the previous method invocation from a JSON string to either an *ExecutableWorkflow* or a *TemplateWorkflow* Java instance, and finally returns this one.

The **saveItem** method firstly checks if the provided Workflow parameter already exists in the database and ontology, and if it is an instance of an executable workflow or a template. Then its behaviour changes according to the previously collected information:

- If the provided parameter is a preexistent workflow, the method updates it in the database by invoking the *editExecutableWorkflow* method of the *DatabaseInterface* class, and in the ontology by invoking the methods *delete-CompositeService* and *createCompositeService* of the *OntologyInquirer* class.
- If it is a nonexistent workflow, the method adds it to the database and the ontology by invoking *editExecutableWorkflow* and *createCompositeService*.
- If it is a preexistent template, the method updates it by invoking the method *editTemplateWorkflow* of the *DatabaseInterface* class.
- If it is a nonexistent template, the method adds it by invoking the method *addTemplateWorkflow* of the *DatabaseInterface* class.

The **deleteItem** method checks the provided Integer parameter to determine if the user is asking for an executable workflow or a template, and invokes either the method *deleteExecutableWorkflow* or the *deleteTemplateWorkflows* one of the *DatabaseInterface* class.

The **executeWorkflow** method simply invokes at first the getItem method of its own class, and then the execute one of the *ExecutableWorkflow* instance found in the previous step, passing the workflow variables provided by the user as input parameters.

9.2.3 Database Interface class

This class is used to directly interact with the workflow database by sending SQL queries to the DBMS.

The class constructor instantiates a connection to the DBMS by getting a data source from the application server JNDI resources, or by using the provided connection string.

Methods **listTemplateWorkflows** and **listExecutableWorkflows** perform a SELECT SQL query asking the names of, respectively, all templates or all executable workflows based on the specified template (if the template parameter is not specified, all executable workflows are retrieved), and populate an *IdDataModel* with the query results.

Methods **getTemplateWorkflow** and **getExecutableWorkflow** perform a SELECT SQL query asking the data field (that contains the workflow JSON string) of, respectively, a template or an executable workflow having the provided name, and return the query results.

Methods **addTemplateWorkflow** and **addExecutableWorkflow** perform an INSERT SQL query that adds to the database respectively, a template or an executable workflow whose data are provided as input parameters.

Methods **editTemplateWorkflow** and **editExecutableWorkflow** perform an UPDATE SQL query that updates information about, respectively, a template or an executable workflow based on data provided as input parameters.

Methods **deleteTemplateWorkflow** and **deleteExecutableWorkflow** perform a DELETE SQL query that deletes from the database, respectively, a template or an executable workflow having the specified name.

9.3 Simple Service Module

SimpleService web service is composed of the following classes, as shown in figure 23:

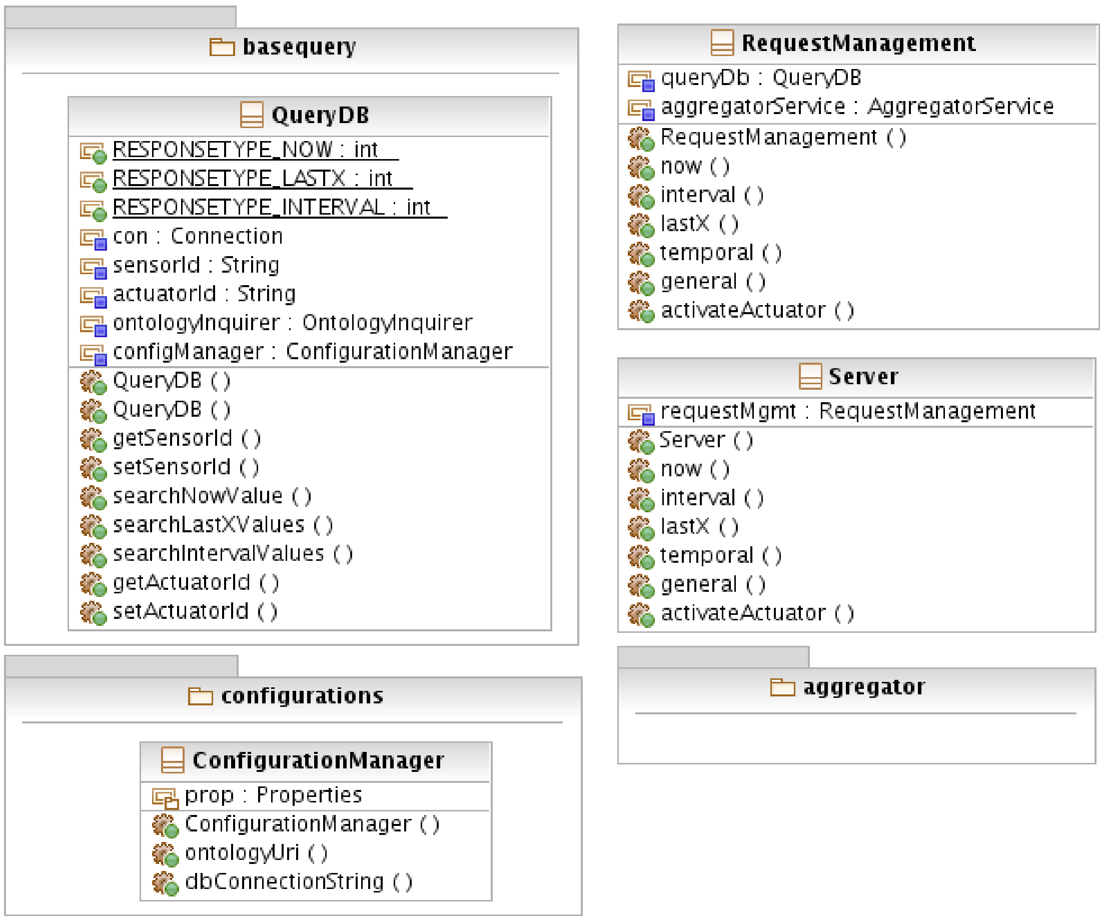


Figure 23: General structure of Simple Service Module

Inside the module there is a ConfigurationManager class, used to read the configuration file and get values associated to some parameters. Every single parameter configure a part of our core functionalities, that the developer can personalize for its needs.

9.3.1 Server class

The Server Class manages the entire REST requests sent by the user. Every request has the same base URL, but appending at the end a specific path causes the invocation of a certain service's method. More in detail:

- For the three types of service (Last One, Last X and Interval, as described in section 6.1.1), it is possible to show particular values registered by the associated sensor.
- In order to operate an aggregation on the data saved by a particular service.
- To activate a particular actuator.

Every method can be configured with dedicated parameters, which can be optional or mandatory. In this way the user can personalize his request. For the complete list of all operation see section B.

In particular:

- For every request, the parameter serviceId is mandatory and represents the identifier of the service that the user wants to access.
- For a Last X request of data it is possible to limit the number of values returned with the parameter sampleNumber. If the service has fewer values, all the data present are returned. If the service has more values, only the most recent ones are returned.
- For an Interval request it is possible to set the temporal span to search particular data, defining the start date and the end date.
- For an aggregation operation it is mandatory to define the type of function that will be applied to the data (for more details see section 9.3.4).
- For an actuator it is possible to also send a parameter defining the desired operation.

All the results are translated into JSON strings.

9.3.2 Request Manager class

The RequestManager analyzes all the parameters sent by the Server Class in order to translate and control the entire request received.

Every method follows the same procedure:

1. At first the received serviceId parameter is controlled: if a service with that identifier exists and has an associated sensor or actuator, the method executes the next step, otherwise it will launch an exception.
2. Then the associated QueryDB or Aggregator method is executed for the requested operation.
3. The final result, a list of values or an exception, is translated in a DataModel format, in order to be translated in JSON by the Server class. For the complete details about DataModel see section 9.4.1.

9.3.3 QueryDB class

QueryDB is the class that manages the interaction with the SCoReS' ontology and the real-time data database. Starting from the requested service, it searches the associated sensor or actuator with **setActuatorId** or **setSensorId**.

With the result found with these methods it is possible to invoke one of the following methods:

- **searchNowValue**, that finds the most recent value registered by the sensor.
- **searchLastXValues**, that returns a list with the most recent values. The size is imposed by the variable *sampleValues* requested by the user and it can be equal or less than that parameter.
- **searchIntervalValues**, that creates a list with the values registered between two dates: *from*, the start date, and *to*, the end date. Both quantities are parameters specified by the user.

9.3.4 Aggregator package

The Aggregator package is used to analyze the data registered by a sensor.

As shown in figure 24, the package is composed by two basic structures:

- A general aggregation, which utilizes an operation on all the data.
- A temporal aggregation, which filters the values with a couple of dates and applies an operation on the subset.

The result is a list that contains the aggregated data.

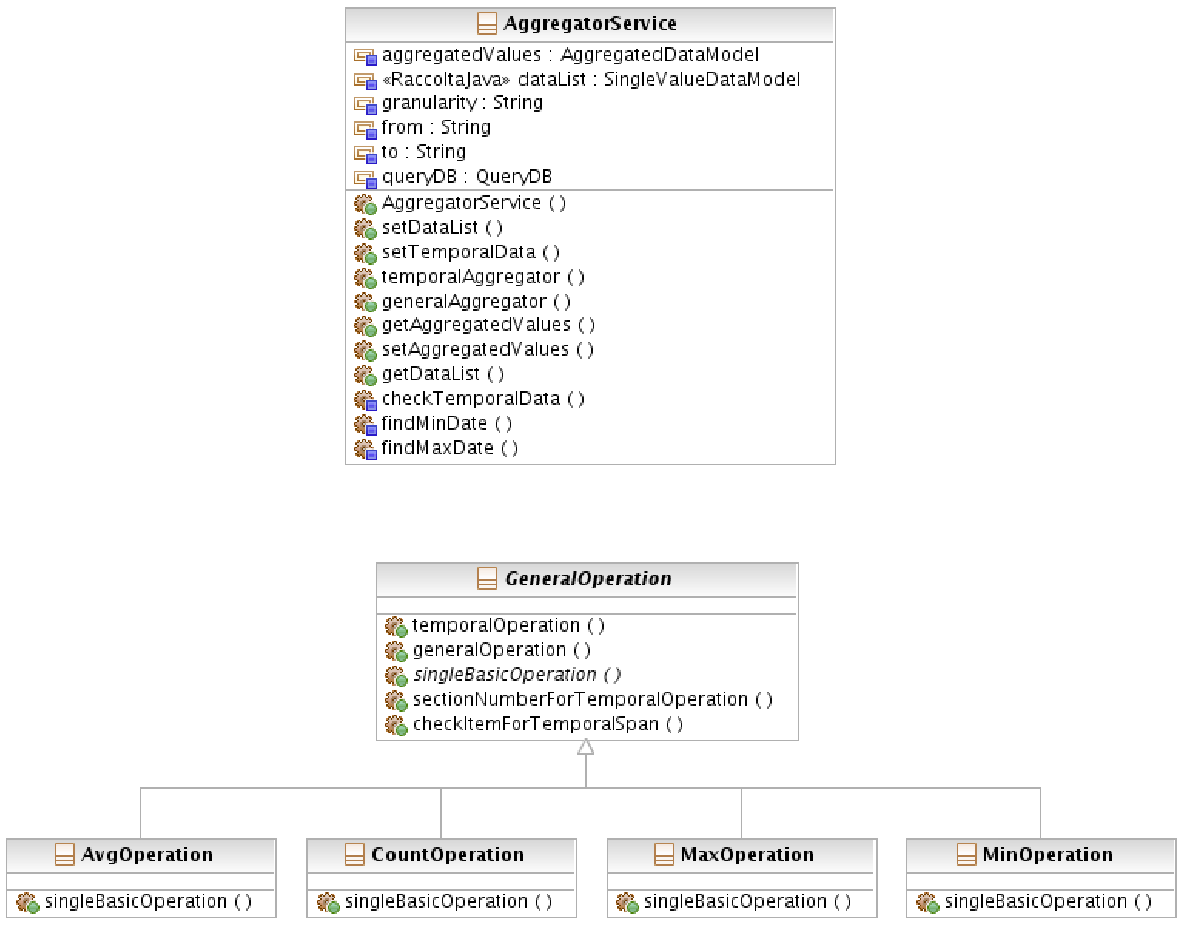


Figure 24: General structure of Aggregator package

A temporal aggregation needs some additional parameter in order to operate:

- A start date and an end date for the considered time interval.
- A granularity, the quantity (expressed in seconds) of the sub-interval that will be used for dividing the list of values.

With the method **checkTemporalData** it is possible to analyze all these parameters and find the missing ones. For example if the start date is not set, the method finds the least recent value registered and imposes the associated date as the start one.

The set of pre-defined operations is composed by:

- **AvgOperation**, which calculates the mean value of a list.

- **CountOperation**, which computes the size of the list.
- **MaxOperation**, which finds the maximum value.
- **MinOperation**, which search the minimum value.

9.4 Shared utilities library

Our shared library is made up of the following elements, as shown in figure 25.

This library contains all the utilities used by the different modules in our project. In addition, it also contains classes that facilitate the integration of our modules in SeNSoRi and the interaction with the graphical interface. As a result the Shared Library is a tool for the programmer, not for the final user.

For a basic interaction, in order to utilize predefined methods in SCoReS, is possible to include:

- **Constant Definition package**, with the explanation and values of the entire set of constant present in SCoReS.
- **Exceptions package**, with the entire set of custom exceptions that can be raised in our project.
- **Formatter package**, with a series of methods that convert strings to dates and vice versa. For a more specific description see section 9.4.4.
- **REST Request package**, with a set of utilities to interact with the Simple Service Module. With the methods contained in this package, it is possible to generate an automatic request to obtain data for a particular service.

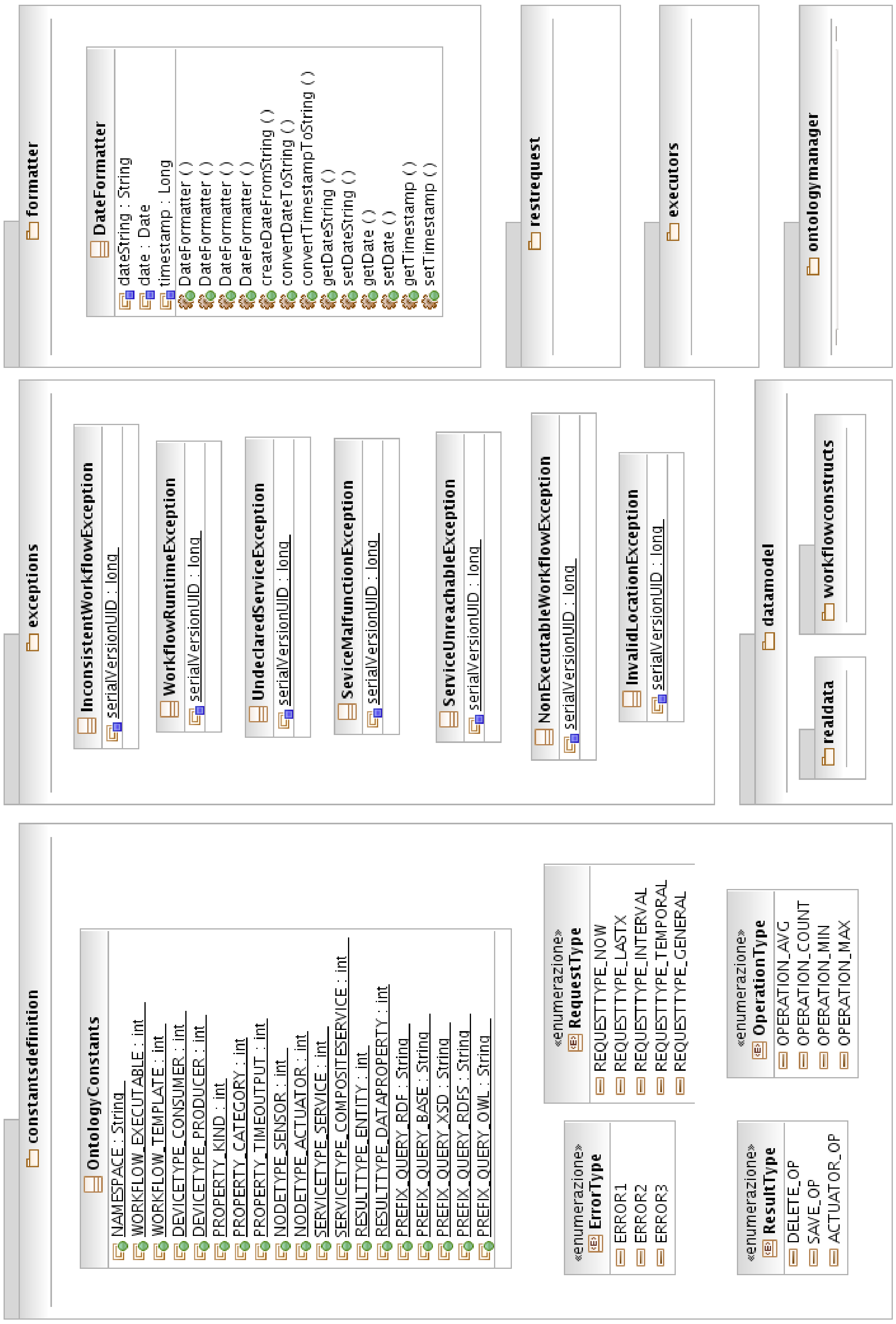


Figure 25: General structure of shared utilities library

9.4.1 Data Model package

In order to have a unified definition of all the possible concepts present in SCoReS, we have created a series of models with special attributes. In figure 26 are shown all the classes.

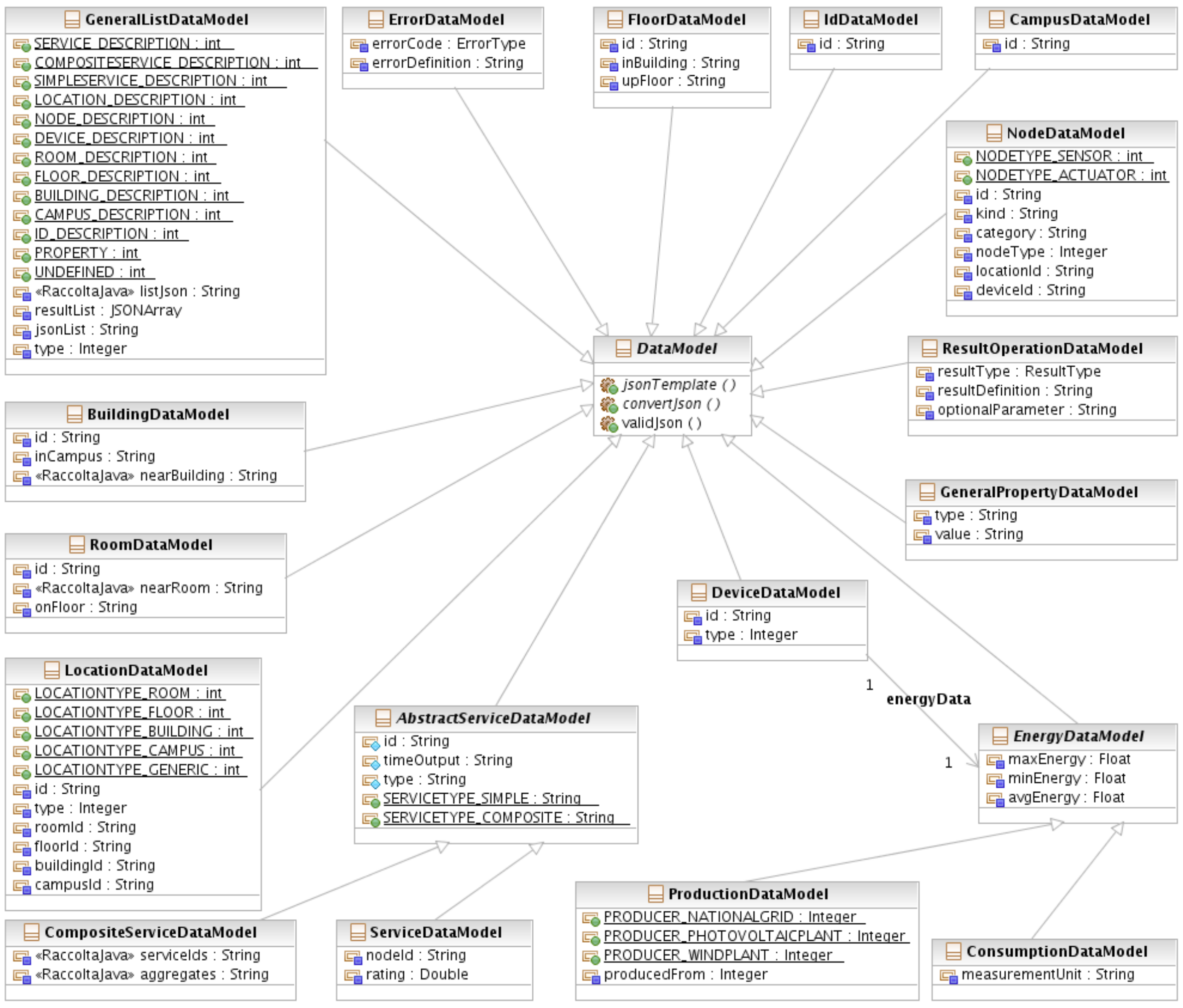


Figure 26: General structure of Data Model package

The main class is **DataModel**, an abstract class that defines the methods that every item must have:

- **jsonTemplate**, which converts the structured data in a JSON object. Every attribute is translated in a JSON format with a couple of parameter names and real values. The real value is extracted from the ontology or the databases present in SCoReS.

- **convertJson**, which receives a JSON definition and converts it in the respective DataModel.
- **validJson**, which controls that a JSON string received is well formatted and doesn't contain an error.

All the remaining classes inherit these methods and, in particular, personalize the jsonTemplate and convertJson ones with their respective set of attributes.

The GeneralListDataModel is the only different method: it converts a list of DataModel in a JSON string and vice versa. In order to be executed successfully it is mandatory that:

- The programmer defines the type of DataModel requested, associating the right constant present in the package.
- A list of JSON definitions or DataModel instances is ready to be analyzed.

9.4.2 Real Data package

Inside SCoReS we utilize a particular set of Data Model classes in order to define the values registered by the sensors associated to a specific service.

There is a predefined hierarchy of classes that compose the final definition of a data set, as shown in figure 27:

1. First a pair made up of a value and its respective timestamp is created, with the data received by the Simple Service Module.
2. Then if the user has requested only a single value, the Data Model created is **SingleValueDataModel**. Otherwise a list with all the results is saved inside **ListNameValuePair**.
3. As last step, the most general Data Model is returned, which represents the requested operation.

In this way it is possible to control the flow of steps and throw an error if some information are missing.

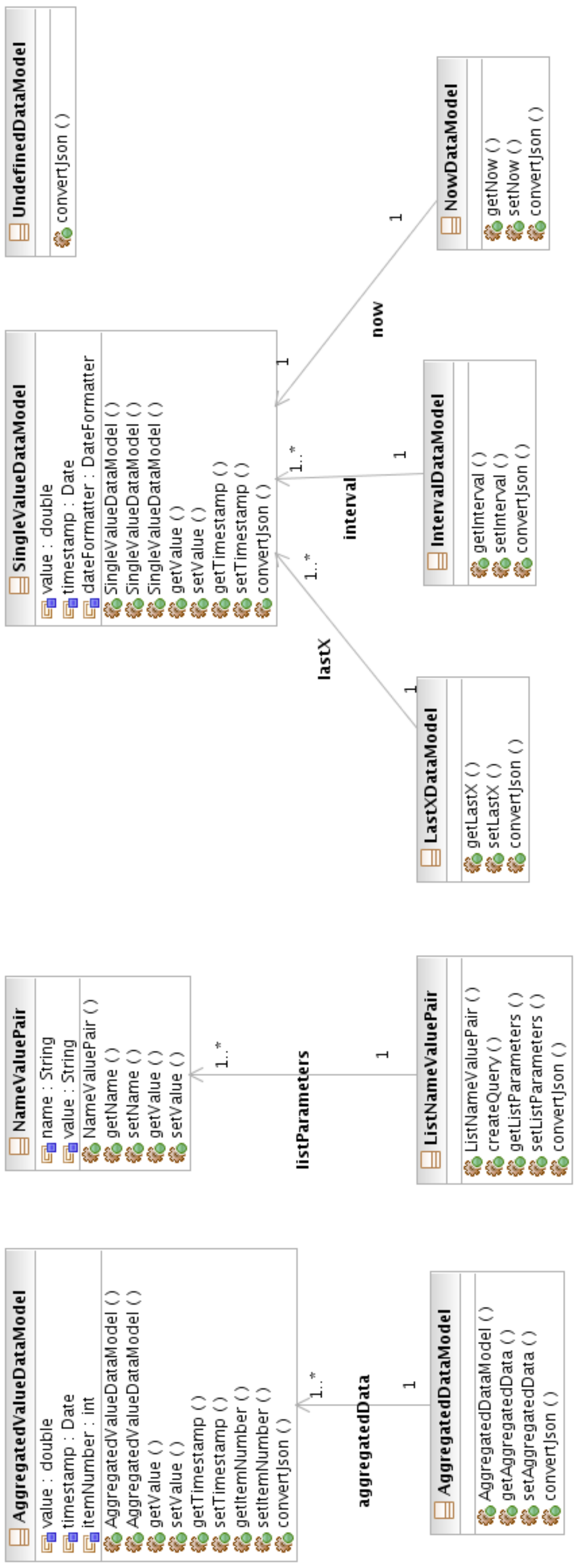


Figure 27: General structure of Real Data package

9.4.3 Workflow Constructs package

This package contains the classes used for representing executable workflows and templates, as shown in figure 28.

These classes and their relationships reflect the structure of our workflow language, as stated in chapter 9.2, with the following differences:

- The distinction between executable workflows and templates is made by an inheritance relationship between the **Workflow** abstract class and the **ExecutableWorkflow** and **TemplateWorkflow** classes.
- The same class (**Invoke**) is used for representing both service invocation and service query workflow elements.
- Instead of containing the service invocation or service query data, the Invoke class contains a reference to either a **Service** class or a **Query** one containing, respectively, all data needed to invoke a service or execute a query that retrieves the desired services.
- It is possible to specify a query either by a SPARQL statement or by providing all constraints by using an instance of, respectively, **SPARQL-Query** or **ParametricQuery** classes, which are descendant of the **Query** abstract class.
- Workflow variables are declared by instantiating the **Variable** class

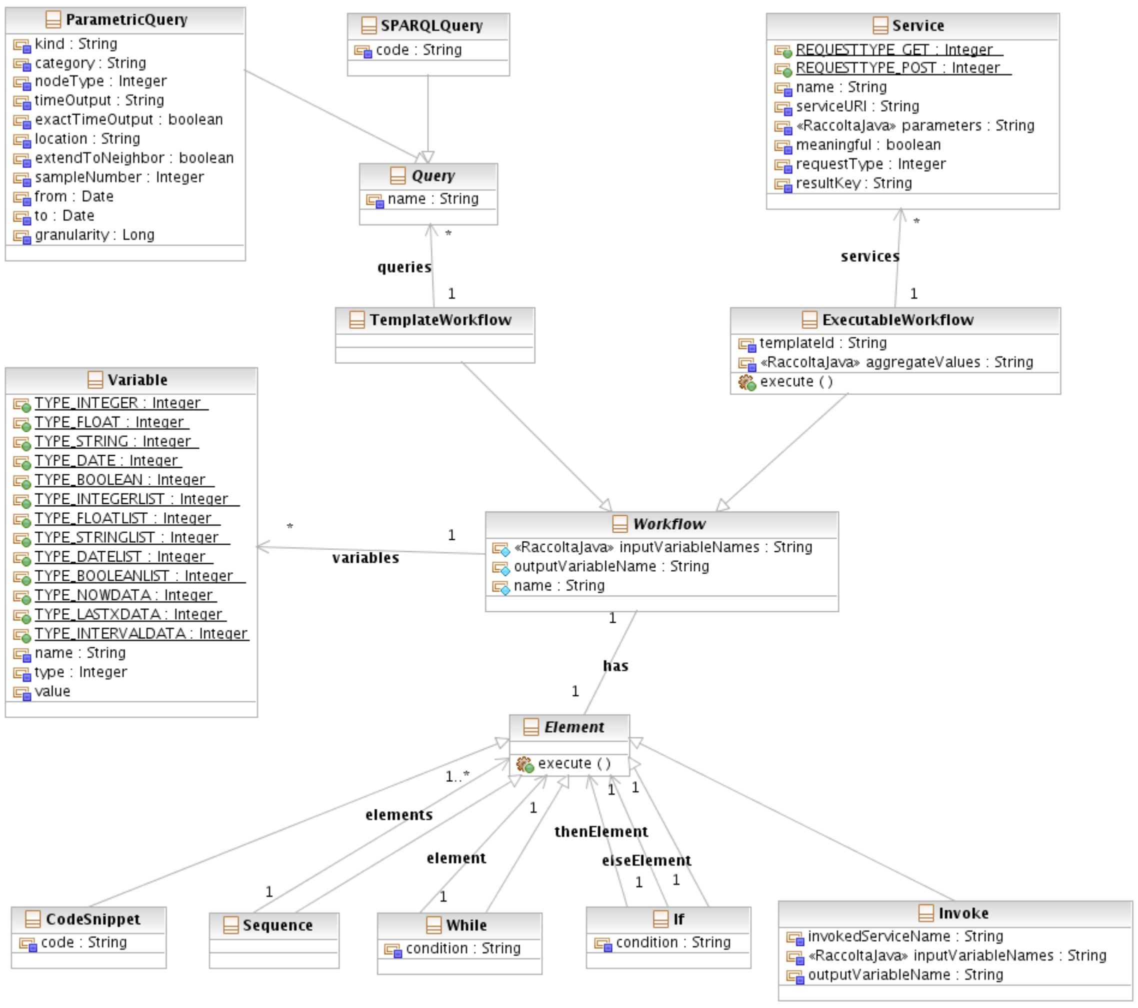


Figure 28: General structure of Workflow Constructs package

Executable workflows are executed by invoking the **execute** method of the **ExecutableWorkflow** class: this method invokes its namesake of the root element, whose behavior depends on the class instance:

- **Sequence** classes invoke in sequence the *execute* method of their referenced elements.
- **While** classes keep invoking the *execute* method of their referenced elements as long as the specified condition is true.
- **If** classes check if the specified condition is true: if so they invoke the *execute* method of the element referenced in the *then* branch, else they invoke the *execute* method of the one referenced in the *else* branch.

- **CodeSnippet** classes instantiate the **ScriptExecutor** class and then invoke the *loadEnvironment*, *executeScript* and *getEnvironment* methods of this one in order to, respectively, load into the engine all workflow variables, execute the code snippet and retrieve the workflow variables modified by the code snippet execution.
- **Invoke** classes instantiate the **ServiceInvoker** class, find the **Service** instance having the same name as the one specified in the **Invoke** one, assign the right variables to the service input and output parameters and finally invoke the *invokeService* method of the **ServiceInvoker** class.

In this way it is possible to execute the workflow elements in a depth-first manner, according to the behavior specified in chapter 9.2.

9.4.4 Formatter package

We have fixed some conventions for defining the specific part of a date:

- yyyy indicates an year with four digits
- MM represent a month with two digits, months between January and September have a leading zero
- dd is the number of a day with two digits, days months 1 January and 9 have a leading zero
- h indicates the hours of a day from 0 to 23, in which the midnight is 0
- m represent the minutes of the day
- s is the second of the specific date
- S represent the millisecond

In order to operate on a specific date, the **DateFormatter** class offers some utilities. Its methods are:

- **convertDateToString**, that converts a specific date to a string with the following format "yyyy-MM-dd h:m:s.S"
- **createDateFromString**, that parses a specific string in its corresponding date. The possible value for the string are "yyyy-MM-dd h:m:s.S", "yyyy-MM-dd h:m:s" or "yyyy-MM-dd"

9.4.5 Executors package

This package contains the following classes:

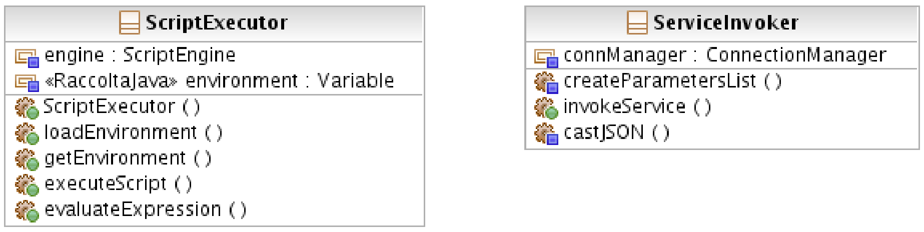


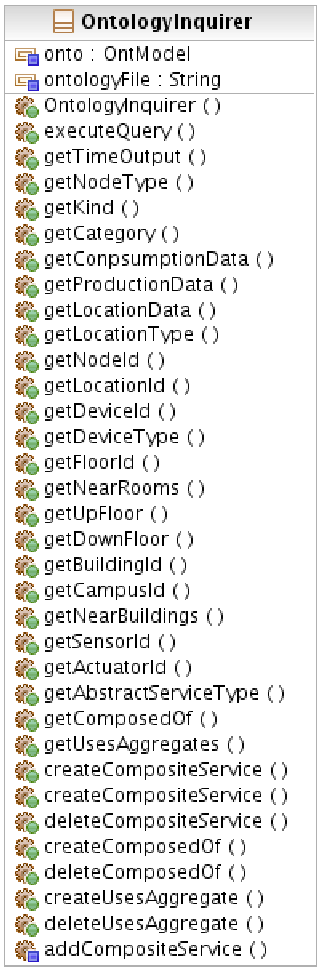
Figure 29: General structure of Executors package

The **ServiceInvoker** class is used to invoke services via the **invokeService** method.
More in detail, this method instantiates the *ConnectionManager* class, builds the parameter name - value list via the *createParametersList* private method and invokes either the *getServerResponse* or postServerResponse method of the *ConnectionManager* class according to the request type specified for the given service.

The **ScriptExecutor** class is used to execute workflow code snippets.
The class constructor instantiates a JavaScript engine.
Methods *setEnvironment* and *getEnvironment* are used to, respectively, assign and retrieve a list of variables to/from the JavaScript engine.
The method *evaluateExpression* is used to evaluate if the given expression is either true or false.
Finally, the method *executeScript* executes the provided code snippet.

9.4.6 Ontology Manager package

This package contains the **OntologyInquirer**, that uses Apache Jena framework [10] in order to query the ontology via SPARQL and to navigate the ontology's structure by returning specific property values for instances that have the given IDs.



```
OntologyInquirer
onto : OntModel
ontologyFile : String
OntologyInquirer ( )
executeQuery ( )
getTimeOutput ( )
getNodeType ( )
getKind ( )
getCategory ( )
getConpsumptionData ( )
getProductionData ( )
getLocationData ( )
getLocationType ( )
getNodeId ( )
getLocationId ( )
getDeviceId ( )
getDeviceType ( )
getFloorId ( )
getNearRooms ( )
getUpFloor ( )
getDownFloor ( )
getBuildingId ( )
getCampusId ( )
getNearBuildings ( )
getSensorId ( )
getActuatorId ( )
getAbstractServiceType ( )
getComposedOf ( )
getUsesAggregates ( )
createCompositeService ( )
createCompositeService ( )
deleteCompositeService ( )
createComposedOf ( )
deleteComposedOf ( )
createUsesAggregate ( )
deleteUsesAggregate ( )
addCompositeService ( )
```

Figure 30: General structure of Ontology Manager package

More in detail, the class constructor invokes Jena methods to load the ontology model, which from now on will be available for the whole class lifecycle.

The **executeQuery** method sends to Jena a user defined SPARQL query, fetches query results (if any) and sends them back to the caller as a list of instance ids.

Methods **getAbstractServiceType**, **getNodeType**, **getDeviceType** and **getLocationType** check if the given instance belongs to a specific class (Service or CompositeService for getAbstractServiceType, Sensor or Actuator for getNodeType, Producer or Consumer for getDeviceType, Location, Room, Floor, Building or Campus for getLocationType) by checking *rdf:type* object property.

Methods **getTimeOutput**, **getKind** and **getCategory** give the value of the *timeOutput*, *kind* and *category* data properties for the given instance.

The method **getNodeId** gives the node id associated to the given service by checking either *dataFrom* or *configures* data property according to the associated node type (sensors have the *dataFrom* property, whereas actuators have the *configures* one).
Similarly, methods **getSensorId** and **getActuatorId** give respectively the sensor id and actuator id associated to the given service by checking either the *dataFrom* or the *configures* data properties.

The method **getConsumptionData** gives the values of the *minConsumption*, *maxConsumption*, *avgConsumption* and *measureUnitConsumption* data properties for the given instance in an aggregated form.
We chose not to have a method for each single data property because we expect the user to be interested only in their aggregated form and not in the single property value.

The method **getProductionData** gives the values of the *minProduction*, *maxProduction* and *avgProduction* data properties and the from object property (specifying if that property relates to a photovoltaic power plant, a wind power plant or the national power grid) in an aggregated form.
Even in this case we chose to aggregate data for the previously mentioned reasons.

Methods **getLocationId**, **getFloorId**, **getUpFloor**, **getNearRooms**, **getBuildingId**, **getCampusId** and **getNearBuildings** give the values of, respectively, *hasLocation*, *onFloor*, *upFloor*, *nearRoom*, *inBuilding*, *inCampus* and *nearBuilding* object properties for the given instance.

The method **getDownFloor** gives the instances having *upFloor* object property equal to the given value.

The method **getDeviceId** either gives the value of the *actsUpon* object property for the given instance if it is an actuator, or gives the id of the device whose influences data property is associated to the given instance if this one is a sensor.

The method **getLocationData** gives the value of the in object property for the given instance limiting the results to the *location* type specified in the method's parameters (room, floor, building or campus).

The method **getComposedOf** gives the service IDs associated to the given composite service by checking the *composedOf* object property.

The method **getUsesAggregate** gives the list of values of the *usesAggregate* data property for the given composite service.

The method **addCompositeService** adds a composite service to the ontology and, if specified, associates it with the provided abstract services via the *composedOf* object property and creates the *usesAggregate* data property values.

The method **deleteCompositeService** deletes the specified composite service. The method **createComposedOf** adds the provided service to the *composedOf* object property of the specified composite service.

The method **deleteComposedOf** removes the provided service from the *composedOf* object property of the specified composite service.

The method **createUsesAggregate** adds the provided value to the *usesAggregate* data property of the specified composite service.

Finally, the method **deleteUsesAggregate** deletes the provided value from the *usesAggregate* data property of the specified composite service.

10 Conclusion and future works

In this document we have presented the SCoReS (SeNSori COmposition & REtrieval Services) project, that offers a retrieval functionality to search information about a service, and a composition functionality to create new services.

The retrieval functionality has been realized by using an ad-hoc similarity algorithm that compares the main aspects in a service, filters out irrelevant services, and returns a ranked list to the user.

The composition functionality, instead, was made available by implementing a custom workflow language and its engine based on JSON, and allows to create new services either from scratch or by using previously defined templates.

At this stage, all SCoReS services have been implemented. Future work will focus on the integration of SCoReS services with the other modules of the SeNSori project, some of which are currently under development, and on the extension of the list of supported workflow patterns (e.g. including the Parallel construct).

A Appendix A

A list of the entire set of JSON message, with examples for every template

A.1 General template

Building Template

Algorithm 9 JSON template for Building concept

```
"id": "Building1",
"inCampus": "Campus1",
"nearBuilding": ["Building2"]
```

Device Template

Algorithm 10 JSON template for Device concept

```
"energyData": {"avgEnergy": 0, "maxEnergy": 700, "measurementUnit":
"W", "minEnergy": 700 },
"id": "TV3D143",
"type": 1,
"typeDescription": "Consumer"
```

Floor Template

Algorithm 11 JSON template for Floor concept

```
"id": "Floor11",
"inBuilding": "Building1",
"upFloor": "Floor12"
```

Location Template

Algorithm 12 JSON template for Location concept

```
"buildingId": "Building1",
"campusId": "Campus1",
"floorId": "Floor32",
"id": "Location97",
"roomId": "Room194"
```

Node Template

Algorithm 13 JSON template for Node concept

```
"category": "Gas",
"deviceId": "KitchenGasOven160",
"id": "Sensor164",
"kind": "Gas",
"locationId": "Location136",
"nodeType": 1,
"nodeTypeDescription": "Sensor"
```

Room template

Algorithm 14 JSON template for Room concept

```
"id": "Room39",
"nearRoom": ["Room40","Room38"],
"onFloor": "Floor5"
```

Service Template

Algorithm 15 JSON template for Service concept

```
"id": "Service439",
"nodeId": "Sensor472",
"ranking" : 1,
"timeOutput": "Now"
"type":"SERVICETYPE_SIMPLE"
```

A.2 Errors list

Algorithm 16 JSON error for invalid request

```
"id": "ERROR1",
"errorDefinition": "Invalid request"
```

Algorithm 17 JSON error for zero results

```
"id": "ERROR2",
"errorDefinition": "Zero results"
```

Algorithm 18 JSON error for no data found

```
"id": "ERROR3",
"errorDefinition": "No data found"
```

B Appendix B

The three main modules developed in SCoReS have a REST interface that allows the user to request data.

The following guidelines explain how to invoke and format every request.

B.1 Service Retriever Module

Base URL: http://localhost:8080/serviceRetrieverImpl/serviceretriever/

Relative path	**Parameters**	**Meaning**
details	*id*=identifier of the desired object *type*=category of the object (for example actuator or sensor)	Request the details of an object present in the system
rooms	*floorId*=identifier of the floor on which search the rooms	Request a list of all the rooms
floors	*buildingId*=identifier of the building inside which search the floors	Request a list of all the floors
buildings	*campusId*=identifier of the campus on which search the buildings	Request a list of all the buildings
campuses		Request a list of all the campuses
sparql	*queryString*=a well formed query in SPARQL *outputType*=the type of the expeted result	Request a particular object from the ontology with a SPARQL query
kinds		Request the list of the entire set of possible value for the parameter kind
categories		Request the list of the entire set of possible value for the parameter category
timeoutputs		Request the list of the entire set of possible value for the parameter timeoutput

Table 6 – continued from previous page

Relative path	**Parameters**	**Meaning**
services	*kind*=value obtained from the REST request *kinds* *category*=value obtained from the REST request *categories* *nodeType*=can assume only two values, one for the sensors the other for the actuators *timeOutput*=value obtained from the REST request *timeoutputs* *exactTimeOutput*=can be true or false, in the second case all types of services are considered *location*=identifier of the location of interest *extendToNeighbor*=can be true or false, in the first case all locations are considered *sampleNumber*=maximum number of registered values for every Last X service *from*=timestamp for the start date, considered only for the the time span for an Interval service *to*=timestamp for the end date, considered only for the the time span for an Interval service *granularity*=time span for the interval described by the parameters *from* and *to*	Request for a list a services with particular characteristics

Table 6: REST invocations for Service Retriever Module

B.2 Workflow Engine Module

Base URL: http://localhost:8080/workflowengine/engine

Relative path	**Parameter**	**Meaning**
retrieve	*type*=can assume only two values, one for the executable workflow the other for the template	Request the entire list of the workflows present in SCoReS
executablewf	*templateId*=identifier of the specific template	Request a list of executable workflows derived from the same template
workflow	*workflowId*=identifier of the workflow to search *type*=can assume only two values, one for the executable workflow the other for the template	Search a specific workflow or template
delete	*workflowId*=identifier of the workflow to delete *type*=can assume only two values, one for the executable workflow the other for the template	Delete a specific workflow [**Mandatory**: DELETE request]
executablewf/save		Save a new executable workflow, in the body of the request is necessary to add all the specification. [**Mandatory**: POST request]
templatewf/save		Save a new template, in the body of the request is necessary to add all the specification. [**Mandatory**: POST request]

Table 7: REST invocations for Workflow Engine Module

B.3 Simple Service Module

Base URL: http://localhost:8080/simpleServiceImpl/serverDB

Relative path	**Parameter**	**Meaning**
now	*serviceId*=identifier of the specific service	Request the last value registered by the specific service
interval	*serviceId*=identifier of the specific service *from*=timestamp for the start date, considered only for the the time span *to*=timestamp for the end date, considered only for the the time span	Request the list of values registered by the specific service between the interval dates
lastx	*serviceId*=identifier of the specific service *sampleNumber*=maximum number of registered values	Request the most recent values registered by the specific service
temporal	*serviceId*=identifier of the specific service *from*=timestamp for the start date, considered only for the the time span *to*=timestamp for the end date, considered only for the the time span *granularity*=time span for the interval described by the parameters *from* and *to* *op*=operation to execute on the data	Request a temporal aggregation on the data registered by the specific service
general	*serviceId*=identifier of the specific service *op*=operation to execute on the data	Request an aggregation on the data registered by the specific service
actuator	*serviceId*=identifier of the specific service *parameter*=value for the activation of the actuator	Request an activation of the specific actuator

Table 8: REST invocations for Simple Service Module

References

[1] Ian F Akyildiz et al. “Wireless sensor networks: a survey”. In: *Computer networks* 38.4 (2002), pp. 393–422.

[2] Sensors Anywhere. 2009. URL: `http://www.sany-ip.eu`.

[3] Michael Athanasopoulos, Kostas Kontogiannis, and Chris Brealey. “Towards an interpretation framework for assessing interface uniformity in REST”. In: *Proceedings of the Second International Workshop on RESTful Design*. WS-REST ’11. Hyderabad, India: ACM, 2011, pp. 47–50. ISBN: 978-1-4503-0623-2. DOI: `10.1145/1967428.1967440`. URL: `http://doi.acm.org/10.1145/1967428.1967440`.

[4] Davide Francesco Barbieri et al. “Querying rdf streams with c-sparql”. In: *ACM SIGMOD Record* 39.1 (2010), pp. 20–26.

[5] D Bianchini, V De Antonellis, and M Melchiori. “Capability matching and similarity reasoning in service discovery”. In: *CAiSE Int. Workshop on Enterprise Modeling and Ontologies for Interoperability, EMOI*. 2005.

[6] Antonio Bucchiarone and Stefania Gnesi. “A Survey on Services Composition Languages and Models”. In: *Proceedings of International Workshop on Web Services Modeling and Testing 2006 (WS-MaTe 2006)*. 2006.

[7] J. Church and A. Motro. “Discovering Service Similarity by Testing”. In: *Services Computing (SCC), 2011 IEEE International Conference on*. 2011.

[8] Nagios Enterprises. URL: `http://www.nagios.org.`.

[9] Federico Fernandez and Jaime Navón. “Towards a Practical Model to Facilitate Reasoning About REST Extensions and Reuse”. In: *Proceedings of the First International Workshop on RESTful Design*. WS-REST ’10. Raleigh, North Carolina: ACM, 2010, pp. 31–38. ISBN: 978-1-60558-959-6. DOI: `10.1145/1798354.1798383`. URL: `http://doi.acm.org/10.1145/1798354.1798383`.

[10] Apache Software Foundation. *Apache Jena*. 2012. URL: `http://jena.apache.org/`.

[11] Akin Gunay and Pinar Yolum. “Structural and Semantic Similarity Metrics for Web Service Matchmaking”. In: *E-Commerce and Web Technologies*. Ed. by Giuseppe Psaila and Roland Wagner. Vol. 4655. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 129–138. URL: `http://dx.doi.org/10.1007/978-3-540-74563-1_13`.

[12] Jeffrey Hau. “A Semantic Similarity Measure for Semantic Web Services”. In: *In: Web Service Semantics Workshop at WWW (2005*. 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.9924.

[13] Antonio Garrote Hernández and María N. Moreno García. “A formal definition of RESTful semantic web services”. In: *Proceedings of the First International Workshop on RESTful Design*. WS-REST ’10. Raleigh, North Carolina: ACM, 2010, pp. 39–45. ISBN: 978-1-60558-959-6. DOI: 10.1145/1798354.1798384. URL: http://doi.acm.org/10.1145/1798354.1798384.

[14] John Ibbotson et al. “Sensors as a service oriented architecture: Middleware for sensor networks”. In: *Intelligent Environments (IE), 2010 Sixth International Conference on*. IEEE. 2010, pp. 209–214.

[15] Younghan Kim. “Restful architecture of wireless sensor network for building management system”. In: *KSII Transactions on Internet and Information Systems (TIIS)* 6.1 (2012), pp. 46–63.

[16] Matthias Klusch, Benedikt Fries, and Katia Sycara. “Automated semantic web service discovery with OWLS-MX”. In: *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. AAMAS ’06. Hakodate, Japan: ACM, 2006, pp. 915–922. ISBN: 1-59593-303-4. URL: http://doi.acm.org/10.1145/1160633.1160796.

[17] Kostis Kyzirakos, Manos Karpathiotakis, and Manolis Koubarakis. “Developing Registries for the Semantic Sensor Web using stRDF and stSPARQL”. In: *International Workshop on Semantic Sensor Networks*. 2010.

[18] F. Lecue et al. “SOA4All: An Innovative Integrated Approach to Services Composition”. In: *Web Services (ICWS), 2010 IEEE International Conference on*. 2010, pp. 58–67. DOI: 10.1109/ICWS.2010.68.

[19] Min Liu et al. “An weighted ontology-based semantic similarity algorithm for web service”. In: *Expert Systems with Applications* 36.10 (2009), pp. 12480 –12490. ISSN: 0957-4174. DOI: http://dx.doi.org/10.1016/j.eswa.2009.04.034. URL: http://www.sciencedirect.com/science/article/pii/S0957417409003741.

[20] Thomas Luckenbach et al. “TinyREST-a protocol for integrating sensor networks into the internet”. In: *Proc. of REALWSN*. Citeseer. 2005.

[21] R. Maigre. “Survey of the Tools for Automating Service Composition”. In: *Web Services (ICWS), 2010 IEEE International Conference on*. 2010, pp. 628–629. DOI: 10.1109/ICWS.2010.72.

[22] Sunil Mamidi, Yu-Han Chang, and Rajiv Maheswaran. "Improving building energy efficiency with a network of sensing, learning and prediction agents". In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems. 2012, pp. 45–52.

[23] Umardand Shripad Manikrao and T. V. Prabhakar. "Dynamic Selection of Web Services with Recommendation System". In: *Proceedings of the International Conference on Next Generation Web Services Practices*. NWESP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 117–. ISBN: 0-7695-2452-4. DOI: `10.1109/NWESP.2005.32`. URL: `http://dx.doi.org/10.1109/NWESP.2005.32`.

[24] Tobias Nestler. "Towards a mashup-driven end-user programming of SOA-based applications". In: *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. iiWAS '08. Linz, Austria: ACM, 2008, pp. 551–554. ISBN: 978-1-60558-349-5. DOI: `10.1145/1497308.1497408`. URL: `http://doi.acm.org/10.1145/1497308.1497408`.

[25] Cesare Pautasso. "BPEL for REST". In: *Business Process Management*. Ed. by Marlon Dumas, Manfred Reichert, and Ming-Chien Shan. Vol. 5240. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 278–293. ISBN: 978-3-540-85757-0. DOI: `10.1007/978-3-540-85758-7_21`. URL: `http://dx.doi.org/10.1007/978-3-540-85758-7_21`.

[26] Cesare Pautasso. "Composing RESTful Services with JOpera". In: *Software Composition*. Ed. by Alexandre Bergel and Johan Fabry. Vol. 5634. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 142–159. ISBN: 978-3-642-02654-6. DOI: `10.1007/978-3-642-02655-3_11`. URL: `http://dx.doi.org/10.1007/978-3-642-02655-3_11`.

[27] Cesare Pautasso. "On Composing RESTful Services". In: *Software Service Engineering*. Ed. by Frank Leymann et al. Dagstuhl Seminar Proceedings 09021. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009. URL: `http://drops.dagstuhl.de/opus/volltexte/2009/2043`.

[28] Cesare Pautasso and Erik Wilde. "RESTful web services: principles, patterns, emerging technologies". In: *Proceedings of the 19th international conference on World wide web*. WWW '10. Raleigh, North Carolina, USA: ACM, 2010, pp. 1359–1360. ISBN: 978-1-60558-799-8. DOI: `10.1145/1772690.1772929`. URL: `http://doi.acm.org/10.1145/1772690.1772929`.

[29] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. "Restful web services vs. "big" web services: making the right architectural decision". In: *Proceedings of the 17th international conference on World Wide Web*. WWW '08. Beijing, China: ACM, 2008, pp. 805–814. ISBN: 978-1-60558-085-2. DOI: 10.1145/1367497.1367606. URL: http://doi.acm.org/10.1145/1367497.1367606.

[30] Mohsen Rouached, Sana Baccar, and Mohamed Abid. "RESTful Sensor Web Enablement Services for Wireless Sensor Networks". In: *Services (SERVICES), 2012 IEEE Eighth World Congress on*. IEEE. 2012, pp. 65–72.

[31] Stefan Schroedl. *Precision-Recall and ROC Curves*. 2008. URL: http://www.mathworks.com/matlabcentral/fileexchange/21528-precision-recall-and-roc-curves.

[32] Amit P. Sheth, Karthik Gomadam, and Jon Lathem. "SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups". In: *IEEE Internet Computing* 11.6 (Nov. 2007), pp. 91–94. ISSN: 1089-7801. DOI: 10.1109/MIC.2007.133. URL: http://dx.doi.org/10.1109/MIC.2007.133.

[33] D. Skoutas et al. "Ranking and Clustering Web Services Using Multicriteria Dominance Relationships". In: *Services Computing, IEEE Transactions on* 3.3 (2010), pp. 163–177. ISSN: 1939-1374. DOI: 10.1109/TSC.2010.14.

[34] Wikipedia. *JSON — Wikipedia, L'enciclopedia libera.* [Online; in data 1-agosto-2013]. 2013. URL: http://it.wikipedia.org/w/index.php?title=JSON&oldid=60017155.

[35] Wikipedia. *Representational State Transfer — Wikipedia, L'enciclopedia libera.* [Online; in data 1-agosto-2013]. 2013. URL: http://it.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=58342996.

[36] Wikipedia. *SOAP — Wikipedia, L'enciclopedia libera.* [Online; in data 1-agosto-2013]. 2013. URL: http://it.wikipedia.org/w/index.php?title=SOAP&oldid=58165108.

[37] Li Zhang et al. "An Approach for Web Service QoS Prediction Based on Service Using Information". In: *Service Sciences (ICSS), 2010 International Conference on*. 2010, pp. 324–328. DOI: 10.1109/ICSS.2010.34.

[38] Haibo Zhao and P. Doshi. "Towards Automated RESTful Web Service Composition". In: *Web Services, 2009. ICWS 2009. IEEE International Conference on*. 2009, pp. 189–196. DOI: 10.1109/ICWS.2009.111.

[39] I. Zuzak and S. Schreier. "ArRESTed Development: Guidelines for Designing REST Frameworks". In: *Internet Computing, IEEE* 16.4 (2012), pp. 26–35. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.60.