Service discovery with personal awareness in smart environments

Kobkaew Opasjumruskit[1], Jesús Expósito[1], Birgitta König-Ries[1], Andreas Nauerz[2], Martin Welsch[1, 2]

Institute for Computer Science, Friedrich-Schiller-University Jena[1]
IBM Germany Research & Development GmbH[2]

Abstract

Web service descriptions with semantic web annotations can be exploited to automate dynamic discovery of services. The approaches introduced there aim at enabling automatic discovery, configuration, and execution of services in dynamic environments. In this chapter, we present the service discovery aspect of MERCURY, a platform for straightforward, user-centric integration and management of heterogeneous devices and services via a web-based interface. In the context of MERCURY, we use service discovery to find appropriate sensors, services, or actuators to perform a certain functionality required within a user-defined scenario, e.g., to obtain the temperature at a certain location, book a table at a restaurant close to the location of all friends involved, etc. A user will specify a service request, which will be fed to a matchmaker, which compares the request to existing service offers and ranks these offers based on how well they match the service request. In contrast to existing work, the service discovery approach we use is geared towards non-IT-savvy end users and is not restricted to single service-description formalism. Moreover, the matchmaking algorithm should be user-aware and environmentally adaptive, e.g. depending on the user's location or surrounding temperature, rather than specific to simple keywords-based search which depends on the users' expertise and mostly requires several tries. Hence, our goal is to develop a service discovery module on top of existing techniques, which shall rank discovered services to serve users' queries according to their personal interests, expertise and current situations.

INTRODUCTION

Context-aware pervasive computing has become a prevalent topic over the last decade. With the upcoming of ubiquitous mobile devices, which provide heterogeneous sensors and facilitate internet access to users, we are close to realising the idea of the "Web of Things" (Mattern & Floerkemeier, 2010). In the Web of Things world, physical things are becoming smart web connected devices. Not only can they create, store and share data, they can even be programmed to make decisions based on these data. In addition to these concepts, things have become not only remotely controllable and able to communicate with each other, but also able to automatically adapt themselves to the demands of the user. By measuring the context of users and preferences, things (in particular resources) can be altered or configured to match the requirements of each user. For example, a user may want to receive an alert, if a weather advisory that concerns him (i.e., refers to a region where he will be when the advisory becomes active) is published. To achieve this functionality, a weather advisory service needs to be combined with information obtained from the user's calendar or his flight reservation service or another service that provides information about his whereabouts.

Even though various works have been devoted to the above concepts, none of them can relieve the painful integration nor ease the complexity of the usage for the non-technical user. Therefore, the major goal of the MERCURY project is to equip the non-IT-savvy users with the user-context adaptive service in order to utilize resources in their own environments. MERCURY is able to support both, generic everyday tasks, like, "set my to-do-task for working out when my daily calorie intake is above the limit", and sophisticated tasks involving several sensors, services and actuators. An example for such a sophisticated task is, "Monitoring medical sensors, such as glucose meter, electrocardiogram (EKG) sensor, respiratory

sensor, etc. When there is any sign of unusual condition, send an alarm to a caregiver or a nearby physician". While maintaining a user-friendly interface, another crucial gap that has not been efficiently addressed in any existing solution is – how to utilize information retrieved from mobile devices or social web in order to provide the user with an automation and recommendation system. Therefore, in order to cover the challenges mentioned above, we continue focusing on the service discovery requirement based on the work proposed in Opasjumruskit, Exposito, Koenig-Ries, Nauerz, and Welsch (2012).

The remainder of this chapter is organized as follows: First, we will review related work, which our assumptions and ideas are based on. We then review the existing approaches for discovering services and carve out the most important aspects of successful and thus most promising solutions. Afterwards, the Background section briefly presents an overview of MERCURY and motivates the relation to the service discovery topic. We present how service discovery fits into MERCURY and provide tangible use cases. Next, we make basic assumptions, which define the scope of our research, elaborate the implementation of the user-context adaptive service recommender. Finally, we describe our proposed architecture.

In section Building Blocks, we explain, in more detail, the request converter along an exemplary use case. In this building block, the service discovery results are improved by involving personal awareness. Next, the result integrator, which is used for merging ranked results from several service matchmakers, is described. We then conclude the chapter and finally, point out the remaining challenges for our future work.

RELATED WORK

Several works have followed the concept of the Internet of things (Mattern and Floerkemeier, 2010), where the Internet extends into the real world embracing physical objects, so that they can be controlled remotely and can be accessed via internet services. For instance, works like SenseWeb (Grosky, Kansal, Nath, Liu, & Zhao, 2007), Web Mashups (Guinard, 2009), SemSorGrid4Env (Gray et al., 2011), Actinium (Kovatsch, Lanter & Duquennoy, 2012) and Bo et al. (2012) enable users to control smart devices through a web-based application. However, most projects are geared towards a few specific use cases and devices. Moreover, the user interfaces presented do usually not allow users to construct a customizable application on their own. Attempts to address such a problem like iCAP (Dey, Sohn, Streng & Kodama, 2006), Phuoc and Hauswirth (2009) aimed towards a non-programming user-defined context sensing application. However, the result is still not sufficiently user friendly and does not support heterogeneity of services. The approach presented by Guo, Zhang, and Imai (2011) supports different types of users, which are categorized by their programming skills. Thus, users can build a complicated application if they are expert users or simple applications if they have no programming skills. However, such user classification is difficult since the definitions are rather diffuse and we believe that the adaptive user interface and the possibility to share one's own applications/scenarios are attractive and tangible in this social network age.

However, we are not the only one who realizes the gap of improving on the internet of things. There are several products already, such as Cloud Business Apps Integration – CloudWork (2013), Cosm - Internet of Things Platform Connecting Devices and Apps for Real-Time Control and Data Storage (2012), Evrythng (2012), IFTTT - If this then that (2013), Paraimpu - The Web of Things is more than Things in the Web (2012), On{X} – automate your life (2012), and the list keeps going on. However, none of them quite meets the requirements outlined above: For Cloudwork and IFTTT, it is obvious that they provide a limited set of web services, which are mostly social network applications. While Cosm, Evrything, Paraimpu and on{X} offer a broader range of sensors and services, they require some programming and hardware skill from users in order to connect building blocks, e.g. sensors and services.

Assuming that systems like MERCURY or the ones described above become successful and that a large number of sensors, services and actuators become accessible via these systems, a major challenge will be to actually find the best fitting sensor, service or actuator among the large number of possible candidates.

In the web service community, several attempts have proposed to make services discoverable. Typically, this happens by embedding a machine interpretable description into the service, based on a domain-specific ontology, as described in Maleshkova, Kopeck, and Pedrinaci (2009). A matchmaking service like for instance, SAWSDL-iMatcher (Wei, Wang T., Wang J., and Bernstein, 2011) and OWLS-MX (Klusch , Kapahnke, and Zinnikus, 2009) can then search for appropriate services. Other courageous approaches using collaborative techniques like WSColab (Gawinecki, Cabri, Paprzycki, and Ganzha, 2010) and Ding, Lei, Yan, Bin, & Lun (2010), also return promising results. Talantikite, Aissani, and Boudjlida (2009) and Martin, Paolucci, and Wagner (2007) discuss all the techniques for semantic annotations of web services, such as OWL-S, WSMO, WSDL-S and SAWSDL. Since the descriptions of web services currently in use are diverse and incompatible, we expect that MERCURY will need to support several of them to achieve best results.

In MERCURY, we have deployed the concept of the Internet of things in the way that users can exploit sensors and devices without programming skills. Even though there are several systems working on the similar direction, we will distinguish our project from the other competitors with a simple integration of sensors and devices. The automatic discovery of sensors with user context awareness could ease users while there are ubiquitous sensors and services spread all over the internet.

BACKGROUND

In this section, we first describe the architecture of MERCURY. Then, we motivate the necessary of automated service discovery we have perceived in our work. Finally, we explain how we use the service discovery in MERCURY.

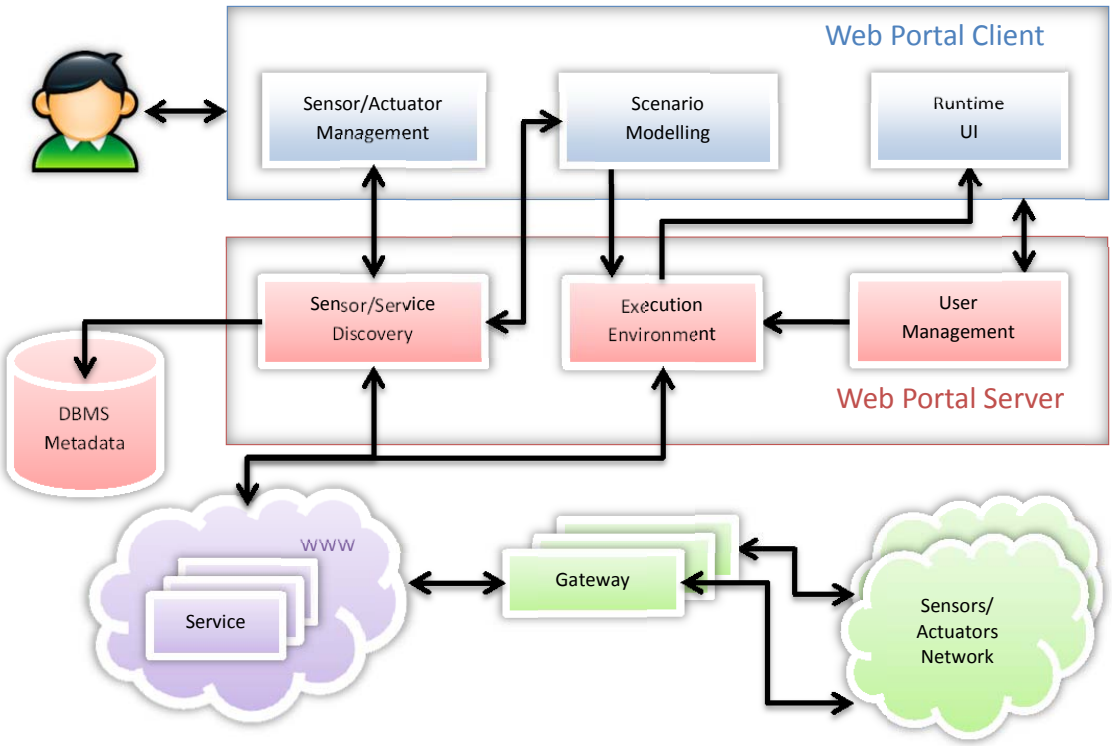


Figure 1 The architecture of MERCURY

MERCURY system architecture

MERCURY offers a service implemented on a WebSphere Portal server (IBM, 2013), which allows users to access MERCURY from any location using any suitable device. In our previous work (Koenig-Ries, Opasjumruskit, Nauerz, and Welsch, 2012) we illustrated the possible architecture of MERCURY, as depicted in Figure 1. On the client side, a *sensor/actuator management* allows the user to add his devices to the system, to publish their descriptions and to define access rights to those devices. After the registration process, a *scenario modelling* allows the user to select and combine devices. Finally, a *runtime UI* will be responsible for displaying events and the status of defined scenarios to the user.

The heart of the system consists of a *sensor/service discovery* module, which manages the sensor's description repository and performs matchmaking of requests from the scenario modelling and descriptions. An *execution environment* obtains the script-like results of the scenario modelling and then accesses sensors, actuators via web services. A *user management* component stores user models (expertise, preferences, etc.) and access rights to devices.

In the bottom part of the architecture, sensors and actuators are either individually or as a network accessible via *gateway* components. These help overcome technical heterogeneity. Gateways are also responsible for bridging sensor networks to web services, so that the sensor/service discovery can access sensors and actuators.

The first prototype (Opasjumruskit et al., 2012) demonstrated that MERCURY could aid the user to accomplish the desired task exploiting environment-adaptive capabilities. For example, a user can assign the location detection service to track her own current position and then automatically choose the appropriate sensor device nearby her, as Figure 2 depicts, and before executing her planned task. In other words, depending on the user's environment or context, MERCURY can perform the assignment with the appropriate selection of resources available at the specific time and place.

(a) (b)

Figure 2 First prototype to demonstrate the environment-adaptive capabilities (a) two sensors are selected and connected (b) the relation between two sensors has been defined

The service offered by MERCURY can be categorized into three main modules: device registration, device management and scenario modelling. The following example can describe the whole usage flow: Imagine an example user, Lea, has decided to go jogging regularly. If there is no rain in the morning, Lea would like to be woken up at 6 A.M. to go jogging. Otherwise, she prefers to receive an alarm message at 7 A.M., and have her calendar updated automatically with an event to go jogging in the evening with her friend instead. Furthermore, Lea works and lives in Stuttgart during the weekdays and spends her time in Freiburg with her family on the weekend. Hence, she would like to maintain the same schedule for jogging at both places. Events and jogging partners will be updated accordingly.

To realize this scenario, a GPS locator, rain sensors close to Lea's living places and her calendar need to be registered via the device registration module. Afterwards, the parameters of these services and devices can be set in the device management module, e.g. update rates or privacy settings. Finally, Lea can wire services and devices to achieve the desired functionality via the scenario modelling module. For the scenario "Jogging plan", she can define several situations, such as, "if it rains" and "set the alarm clock" situations, so that the user can alter and reuse the scenario easily. A situation like "if it rains" can be composed of a GPS locator and a weather forecast service. On the other hand, an "if it does not rain" situation can be realized by adding the "if it rains" situation and a negation operation to the scenario. Each situation can be saved as a template for later use, the template can be shared among users and, eventually, each situation can be altered individually.

As described above, MERCURY allows flexibly combining sensors, actuators, and services to model a desired functionality. For the work described in this chapter, we assume that sensors and actuators are wrapped as services, so that everything can be accessed via web service interfaces. In order to achieve maximum flexibility, we do not make any assumptions whether a given service wraps one individual sensor, an entire sensor network, or some combination of sensors and actuators.

Automated Service discovery

In order to facilitate the user in searching for a service she needs, the automated service discovery plays an important role here. The basic idea behind the automated service discovery is to annotate services with machine interpretable descriptions. Usually, when a human user or an application is looking for a service she will provide a service request in a predefined format specifying the capabilities of the service she is interested in. A matchmaking component then compares available service descriptions and the request and returns the service(s) that best match the user's request.

For our work, we assume that the minimum information available for any web service is its WSDL description. The Web Services Description Language (WSDL) (Christensen, Curbera, Meredith, and Weerawarana, 2001), a World Wide Web Consortium (W3C) recommendation, is a syntactic description of a web service's interface. In our context, however, a WSDL description alone is not helpful, since it is meant mostly for human consumption and does not contain sufficient information for automated service discovery. The latter aim was first addressed by the AI (Artificial Intelligence) community which developed a number of ontology based techniques, such as OWL-S (Web Ontology Language for Web Services) (Martin et al., 2004) and WSMO (Web Service Modelling Ontology) (Roman et al., 2005). These rather heavyweight approaches describe a service's capability in terms of its preconditions and effects in addition to the service's interface. The descriptions use powerful logic languages, which allows for the application of reasoning techniques. However, providing these descriptions requires significant efforts. Thus, up to now, uptake outside of research projects has been rather slow.

In an attempt to ease the way into semantic service descriptions, SAWSDL (Semantic Annotations for WSDL) was created. SAWSDL allows for semantic annotations of WSDL descriptions using arbitrary ontologies (Hobold and Siqueira, 2012). SAWSDL is a hybrid description that provides both syntactic and semantic discovery of services. Another lightweight option to support service discovery is the usage of community-provided tags as described, e.g., in Gawinecki, Cabri, Paprzycki, and Ganzha (2012).

For all of these service description approaches mentioned above, different matchmakers have been proposed, as Tran, Puntheeranurak, and Tsuji (2009) and Ngan, Kirchberg, and Kanagasabai (2010) have reviewed. There exist several initiatives to evaluate the respective merits of the different description formalisms and matchmaking algorithms, among them, according to Blake, Cabral, König-Ries, Küster, and Martin (2012), are the Semantic Web Services Challenge (Harth and Maynard, 2012), the Web Service Challenge (Blake, Bleul, Weise, Bansal, and Kona, 2009) and the S3 Contest (Klusch, 2012).

Since we want to be able to integrate arbitrary services available over the Internet, we cannot make any assumptions about the description framework used. Therefore, in our solution, we try to support several

different description formalisms, including OWL-S, SAWSDL, and tag-based. For matchmaking, we will rely on those matchmakers that the S3 contest has ranked best.

In order to exploit existing service descriptions and to perform matchmaking as described above, appropriate service requests are needed. Quite obviously, ordinary users will not be able to provide such requests in one of the formal languages. Therefore, we rely on an approach similar to the one proposed in WSColab (Gawinecki et al., 2010), a tagging based solution: Gawinecki (2009) proposed the collaborative tagging with structure technique as a solution. To create a collaborative tagging description, we can use a structure based on WSColab system tags (Gawinecki, et al., 2012).

Crucial information about the service descriptions has been categorized into three categories: input (information the service needs), output (information the service provides), and behavior tags (description of the operation of the service). Afterwards, the request message will be converted to WSDL, SAWSDL, OWL or tagging format and be forwarded to the matchmaker module. The in-depth detail and example for the conversion are explained in the Building Blocks section.

In addition to the service discovery aspect, user profiles and context should be taken into account. Soylu, Causmaecker, and Desmet (2009) presented in the context of Pervasive Computing the idea to react to environmental conditions and adapt accordingly. However, automatic actions can easily lead to undesired results. Thus, users should involve resolving conflicts of priorities or desired behavior MERCURY should only suggest appropriate and compatible services relevant in the given context. This way the user can make decisions easier without being overwhelmed by numerous irrelevant services. Additionally, Breslin, Passant, and Decker (2009) presents the idea to derive user context from their social activities, e.g. from their interest, their expertise and their friend's activities. They consider social networking activities as an important aspect of user context.

In MERCURY, we define the situation as an event to identify specific outcome from sensor(s), or specific action to actuator(s), while the scenario performs as a whole application that is composed of several situations. Therefore, each situation is reusable and each scenario is easier to customize. We envision four main use cases of the service discovery module in MERCURY as shown in Figure 3: Service Registration, Situation definition, scenario definition, and scenario execution.

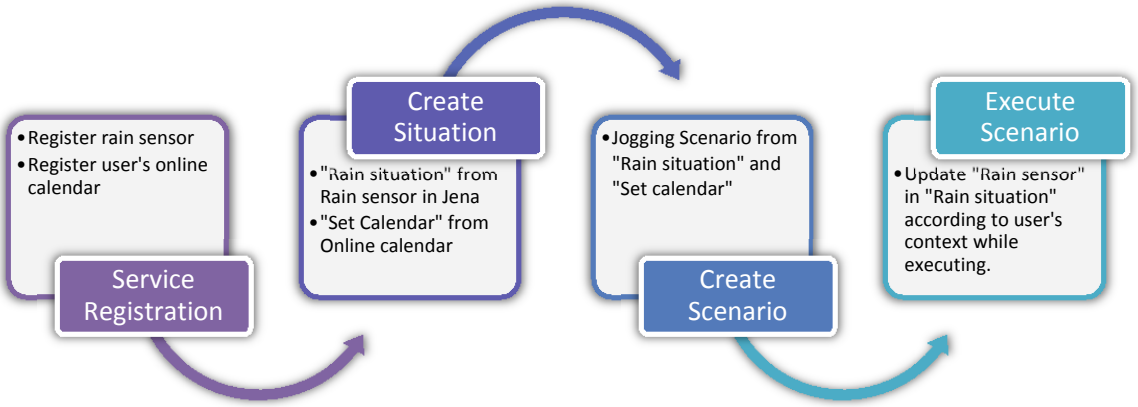


Figure 3 Process flow of Sensor discovery in MERCURY

Usage of discovery in MERCURY

1. Service Registration: Before a user can utilize sensors, actuators, or services in her situations or scenarios, she needs to register them with MERCURY. While this approach makes sense, e.g., it is not realistic to register all of the potentially relevant services available via the Internet. Therefore, we complement the manual registration with an automated service discovery approach.

Error! Reference source not found. Figure 4 shows how the automated service discovery performs in service registration.

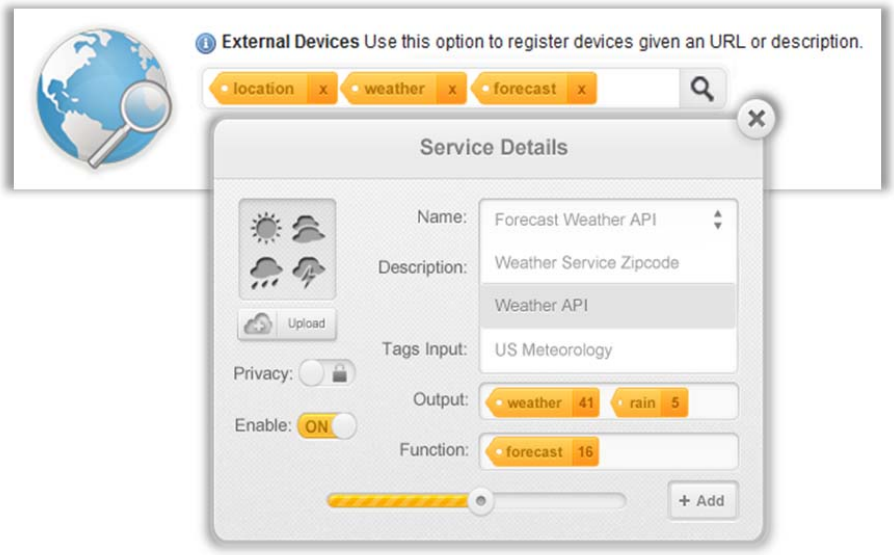


Figure 4 The automated service discovery offered in service registration module of MERCURY

2. Situation Definition (create): When a user creates a new situation, either from scratch or from a template, she might use the discovery module to find appropriate services to add to this situation. Consider, e.g., a user defining a "Rain in Jena" situation looking for an appropriate sensor to implement this situation.

   We can create a situation by two approaches.

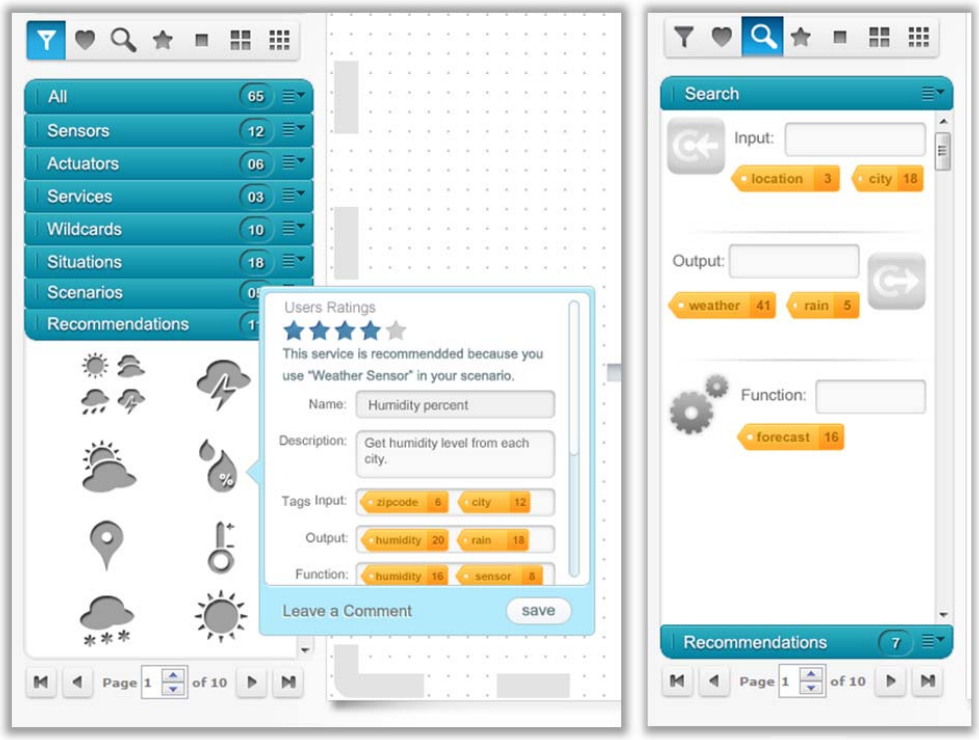
   - *From scratch* -> User searches for sensors one by one and adds them to a situation.
   - *From existing situation* -> a previously created situation can be treated as a template. The description available for each situation can be defined by a user or automatically created from sensors' descriptions used in that situation.

   Thus, service discovery plays a role as a sensor or service search engine. A user can search for existing services in order to model a situation. For example, he could use the service discovery module to discover weather sensors and sprinkler actuators in his backyard.

   A service control panel in the Mercury modelling module is shown in Figure 5. There are two ways of discovery here: implicitly and explicitly. Figure 5(a) shows MERCURY implicitly recommends situations. Users can create, edit or delete situations easily via a control panel. Once a service has been added to the situation canvas, a service request is created implicitly and the resulting services are put in the recommendation panel. From Schafer, Frankowski, Herlocker, and Sen (2007), the recommendation can be based on collaborative filtering, such as rating from other users who have the same behavior, or based on content-based filtering, which does not need other users' efforts, but rather depends on services' descriptions.

   Apart from the implicit recommendation, MERCURY also provides users an advanced search function, where a service request is created from explicit user inputs. Users can search for the service with specific inputs, outputs and functionalities as shown in Figure 5 (b). Moreover, if a service description contains semantic annotations; more service that is relevant can be discovered.

After all services have been connected, the situation can be saved and the user will be prompted to specify a name and a description of this situation. From the user's definition of the description, MERCURY will create a description file (like SAWSDL or collaborative tagging, for example) so that the service matcher can discover the situation later.



(a) (b)

Figure 5 Prototype for Modelling Module searching for a service or sensor (a) service recommendation (b) advance search in modelling module

3. Scenario Definition (create): Similarly, when a user creates a new scenario from existing situations, she may want to use the discovery module to find appropriate situations to include in the scenario. The discovery process can be done implicitly based on the user's contexts or explicitly from the user's inputs as in the situation definition. Consider, as an example, a user aiming to define: if "Rain in Jena" then "set my calendar later as - I won't go jogging". Hence, she might look for a previously defined "Rain in Jena" situation.

   Figure 6 displays an example in which a user created the situation called "Is it raining?" It will notify the user if a precipitation phenomenon occurs during a specific time frame of a day. The situation is triggered once it is raining or snowing at this time, let say early in the morning. Additionally, the situation can be tagged, shared and rated for the sake of community distribution and recommendation.

   The situation template can be selected and edited to serve a particular user's need. For example, a user can select an existing template named "Turn on the light, if anyone enters the room". He can edit a parameter in the situation from "anyone" to "me" by changing the movement detector service to an RFID service that identifies a person who enters the room. To achieve this, the

sensor discovery module will play two roles: discovering an existing situation and discovering an appropriate service.

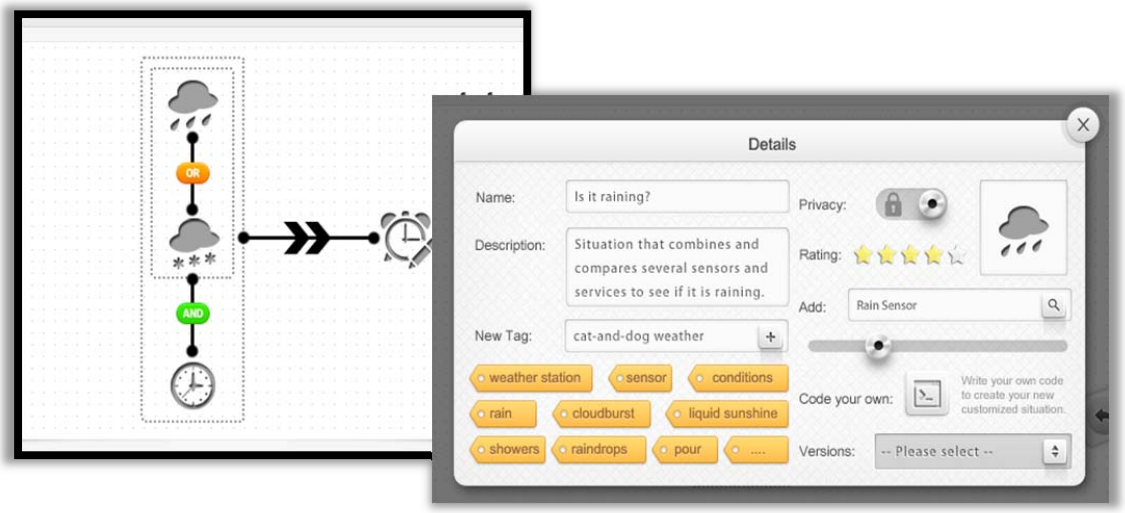


Figure 6 Prototype for Modelling Module creating the situation

4. Scenario Execution (dynamic discovery of sensors): Users may also want to define generic situations that are automatically instantiated with appropriate services at runtime. Consider, e.g., a user who travels a lot. She might decide to replace the condition in the scenario above by a “Rain where I am” situation. Assume further, that for some reason she does not trust the web-based weather services, but would like to rely on readings of “real”, physical sensors in her vicinity. In this case, the situation would need to be changed, so that, first, a location sensor (maybe the GPS sensor in her cell phone) determines her current location. Next, a service request enables the run-time discovery of a rain sensor close to that location, which resulting in the discovery, configuration, and ultimately execution of the service.

   For standard scenarios, there is no need to re-discover the service again. However, if the user needs to change the service automatically respect to his or her current context, Mercury needs to use the “on-the-fly” service discovery. For instance, we have a situation says "Detect if there is a post office nearby my current location".

   From the sensor discovery, the user selected a GPS sensor to provide a parameter "mylocation". During an execution of this situation, the rain sensor service should be re-run according to the value from “mylocation”. At runtime, MERCURY will construct a request message for dynamic discovery of the service. The parameter “mylocation” is used as a keyword in service behavior, a service input is left blank and a service output can be an address of the nearest post office with approximate distance.

There is a significant difference between cases 1, 2 and 3 on the one hand and case 4 on the other hand: Case 1, 2 and 3 are user-interactive. MERCURY will propose suitable sensors, but it is up to the user to choose the most appropriate one. In the fourth case, such interaction is hardly possible, since it is highly doubtful that the user would like to be awakened by MERCURY at 5 a.m. to decide whether the chosen rain sensor meets her requirements. We thus need to ensure that MERCURY is capable of automatically finding the best sensor.

SERVICE DISCOVERY TECHNIQUES

In this section, we describe how we envision service discovery to work in MERCURY and how the discovery technique can be made aware of personal contexts and preferences in order to provide the proper recommendation or desirable result. We start with some assumptions to define the scope of this topic. Then, we present an architectural model of service discovery that can fulfil the previously mentioned requirements.

Initial assumption

In MERCURY, as briefly mentioned above already, we assume that all sensors and actuators are available as web services. We do not make any assumptions whether a web service represents a single sensor/actuator or provides access to an entire sensor net. We do assume that all these web services are described using WSDL. In addition to these WSDL descriptions, which are necessary for the invocation of the services, the services may possess additional descriptions. These additional descriptions may be, e.g., OWL-S, SAWSDL, hRESTS or Micro-WSMO and/or "social" descriptions gained from user-provided tags. MERCURY itself does not support semantic annotation of services. It does provide an interface for tagging, though.

It is compulsory to specify the domain of services we can discover here, since the ontology repository needs to be defined. We also assume that the installation of MERCURY will be supported by a certain business purpose, such as insurance or logistic industries, so that MERCURY can assign the ontology to a request from a user automatically. Otherwise, the request that contains semantically ambiguous descriptions, for example, "apple", which can be defined in the domain of fruit or brand, can lead to wrong analysis results.

Architecture

Figure 7 shows the architecture of the MERCURY service discovery module. First, a request needs to be created. To achieve this, the user needs to give the input via the Request UI, which requires input keywords or output keywords or service behavior keywords (Mercury needs at least one keyword, it is not necessary to provide them all. The more keywords are provided, the higher the matching quality will potentially be). The same methodology is used to support automatic filter, i.e. proposing only matching components when working with the modelling UI. In this case, the input is derived from state and context of the modelling module depending on the current content of the scenario.

Imagine a user who needs to look up for the weather forecast service, which is indicated by zip code or city name. In this case, the service that the user is looking for should accept zip code or city name as inputs, return weather details as outputs, and, optionally, the behavior (in the other name, operation description) of the service can be described as "weatherByZipcode" or relevant terms (such as "cityWeatherForecast", etc.). Therefore, we can construct a request message as depicted in Table 1.

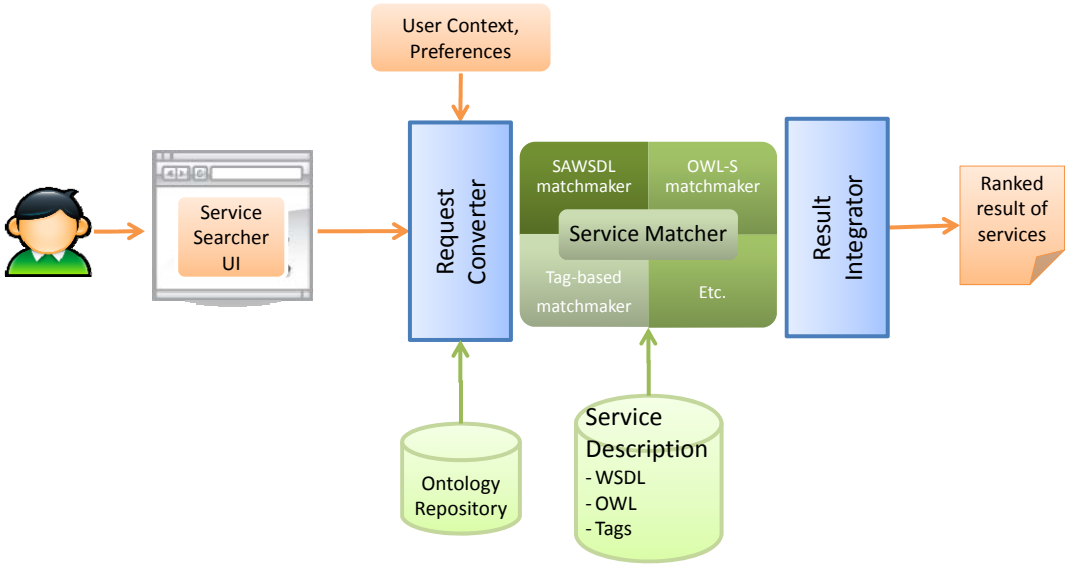


Figure 7 MERCURY Service Discovery Architecture

```
<request>
    <message name=”input”>
        <element name=”zipCode” type=”string”/>
        <element name=”cityName” type=”string”/>
    </message>
    <message name=”output”>
        <complexElement name=”weatherForecast” type=”string”>
            <subElement name = “date"/>
            <subElement name = “weatherDetail"/>
        </complexElement>
    </message>
    <message name=”behaviour”>
        <element name=”weatherByZipcode” type=”string”/>
        <element name=”weatherByCityName” type=”string”/>
    </message>
</request>
```

Table 1 Service request formatted message constructed in MERCURY

Optionally, input and output can be restricted to terms from a controlled vocabulary or ontology. This will improve match results. We will explain the technical details underlying the request converter in section Building Blocks. From these user inputs and possibly user context or preferences, service requests are created. Since we do not know which description formalisms were used by the service providers, we convert the user input into requests following different formalisms. For the proof-of-concept implementation, these will be SAWSDL, OWL-S (input/output only), and tag-based following the model proposed by WSColab (Gawinecki, 2009).

Table 2 shows the example of a service description in SAWSDL format. It was constructed from WSDL format, which contain service’s information those are necessary for automatic service discovery, such as

names, binding methods, inputs and outputs of a service. In addition to the basic description, the semantic annotations (notate in "sawsdl:modelReference") were added so that the machine can exploit more from the description. Considering that, the input element name "city" without semantic annotation cannot be referred to its super classes, such as "place" or "administrative area".

```
<wsdl:description …>
    <wsdl:types> <xsd:schema …>
            <xsd:element name=”WeatherServiceRequest” type=”xsd:string”
            sawsdl:modelReference=”http://.../ontosem.owl#city”/>
            <xsd:complexType name=”WeatherServiceResponse”
            sawsdl:modelReference=”http://.../ontosem.owl#weather”>
                <xsd:sequence>
                    <xsd:element name=”date” type=”Date”/>
                    …
                    <xsd:element name=”weatherDesc” type=”String”/>
                </xsd:sequence>
            </xsd:complexType>
    </xsd:schema> </wsdl:types>
    …
    <wsdl:interface name=”WeatherServiceInterface”>
        <wsdl:operation name=”WeatherService”>
            <wsdl:input element=”WeatherServiceRequest” />
            <wsdl:output element=”WeatherServiceResponse” />
        </wsdl:operation>
    </wsdl:interface>
    <wsdl:service name=”WeatherService” >…</wsdl:service>
</wsdl:description>
```

Table 2 Service description in WSDL format, enhanced with semantic annotations (SAWSDL)

The example for an OWL-S service description is represented in Table 3. Apart from the difference of languages, its basic construction is similar to SAWSDL in that it contains service name, inputs and outputs. While OWL-S allows for the specification of preconditions and effects, none of the current matchmakers takes advantage of this information; we thus do not make any effort to provide this information.

```
<rdf:RDF  …>
    <owl:Ontology rdf:about="">…</owl:Ontology>
    <service:Service rdf:ID="WEATHER_FORECAST_SERVICE">…</service:Service>

    <process:Input rdf:ID="_CITY">
        <process:parameterType> http://.../ontosem.owl#city</process:parameterType>
    </process:Input>

    <process:Output  rdf:ID="_WEATHER">
        <process:parameterType> http://.../ontosem.owl#weather</process:parameterType>
    </process:Output>
    …
</rdf:RDF>
```

Table 3 Service description in OWL-S format

<service name="Weather_Forcast_Service.xml"> <behaviour> <annotation users="3" tag="weather_forecast"/> <annotation users="2" tag="city_weather"/> <annotation users="2" tag="weather_service"/> </behaviour> <input> <annotation users="4" tag="city_name"/> <annotation users="1" tag="city"/> </input> <output> <annotation users="3" tag="weather_condition"/> <annotation users="2" tag="temperature"/> <annotation users="1" tag="weather"/> </output> </service>	input:city output:weather behaviour:weather_forecast_service
(a)	(b)

Table 4 (a) Service description and
(b) service request format for collaborative tag-based method described in WSColab

The tag-based service description in Table 4 (a) exemplifies how the collaborative tagging is stored. Each service contains tags for behavior, input and output of the service. In addition, each tag contains the number of users who tagged the service with the same keyword. Table 4 (b) shows the example of service request that can be constructed from message in Table 1.

We will use existing service matchers, such as, SAWSDL-iMatcher (Wei, Wang, Wang, & Bernstein, 2011), OWLS-MX (Klusch, 2009), etc. Each service may have more than one type of description (e.g. SAWSDL and tagging). In such a case, the user would not need to specify the type of description for the service he or she is looking for. The matcher will compare the request message with description files, which are already known by matchers.

Finally, the results returned from each matcher will be merged and rearranged again using weight parameters, which will be described in more details in section Building Blocks. The weight parameters are configurable and can be determined as described by Klusch (2012). Match results from matchmakers with good evaluation results will be weighed more heavily than those from matchmakers with poorer performance.

The service discovery module acquires user's request from a text-based search, and separated between operation, input and output terms, and converts them into formatted structures. There is no specific service matching technique for our approach, since each service can be described in different forms. We also cope with all possible (and prominent) matching techniques by converting the request into required formats. Finally, the results from each service matchmaker are needed to be sensibly ranked. Therefore, we will discuss the necessary building blocks further in detail.

BUILDING BLOCKS

From the architecture of the MERCURY service discovery module, the two main building blocks, here we describe the request converter and the result integrator. The request converter section shows how the service request works with service matchmakers, which have various formats.

The result integrator section explains how MERCURY can rank the returned result from the service matchmakers, according to the relevance, and how the user context can manipulate the recommendations of services.

Request Converter

After a user submits a request, this data should be converted into an appropriate form for service matchers. As shown in Figure 8, the request converter will retrieve information from a configuration file for the name and the directory of service matchers, so it can execute them. The request format is the key parameter for the request converter, since the request will be reformatted again according to this information. In addition to the service description, MERCURY will enhance it with semantic annotation. Thus, text analysis within the specific domain should automatically assign the ontology to the request. Currently, our implementation is using OWL ontologies following the test collection in Klusch (2012). The weighted index will be used in the result integrator, which is described in the next section.

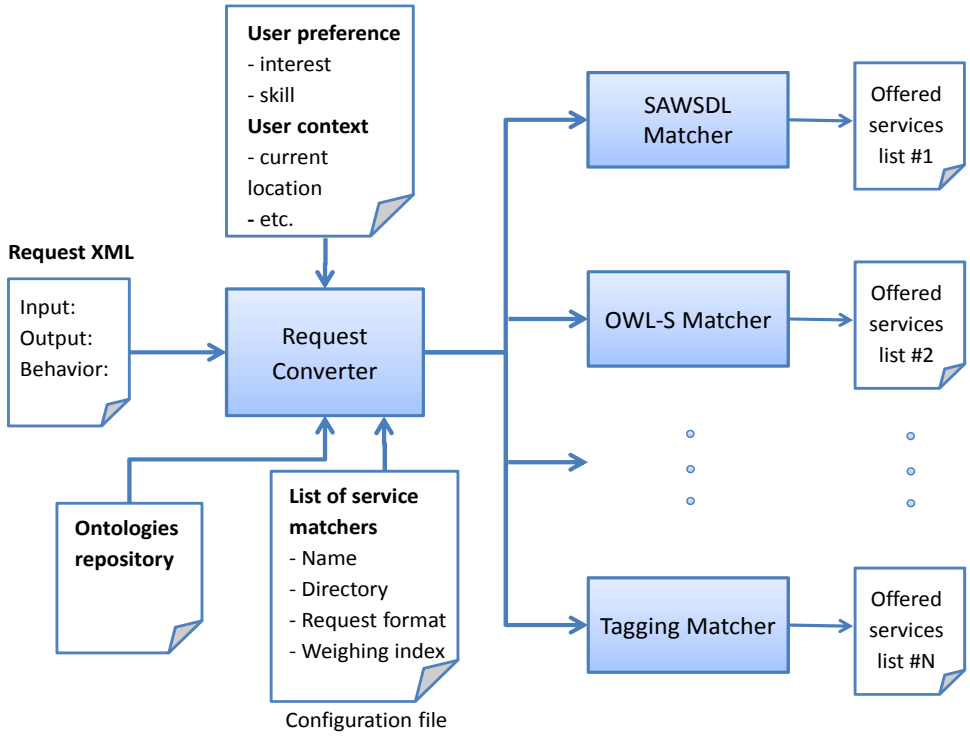


Figure 8 Data flow diagram of Request Converter and Service Matcher

A request message is converted to a compatible format of each matcher following a mapping schema. For example, to convert a request message shown in Table 1, the request converter will look for a mapping schema for OWL-S, which is specified in a configuration file. The mapping schema, as Table 5 shows, will specify the element and attribute of the request message to OWL-S format. Additionally, the semantic annotations of the input and output context can be automatically defined before the conversion. The request converter will look up for the semantic description of the word "weather" in an ontology repository. The result will be returned as URIs, which is defined as ontology attribute of an element.

In addition, user preference and user context are optional inputs, whenever they relate to the service being requested. Otherwise, this information will not be added to the request.

To exemplify how the request would look like before being processed by the service matchmaker, we use the SAWSDL based matchmaker as an example, as presented by Klusch (2009). Please note that we have omitted some information in the service request for the sake of simplicity. For other types of matchers, the same concept is applied.

```
[input]
        element(name) -> process:Input (rdf:ID)
        element(ontology) -> process:parameterType (node value)
[output]
        element(name) -> process:Output (rdf:ID)
        … // in this example, this part is similar to the input
[behaviour.request]
        element(name) -> grounding:wsdlInputMessage (node value)
[behaviour.response]
        element(name) -> grounding:wsdlOutputMessage (node value)
```

Table 5 Mapping schema example for OWL-S

From Table 6, the request for a service is matched with the offers in the service repository. As an example, the service request defines that the required service should provide two types of operations, both offerthe weather forecast description as outputs, but one accepts a zip code as an input, and the other accepts a city name as an input. The first service offer provides both operations. Even though it contains different keywords from the request, the vocabulary problem can be resolved with the semantic annotations. The second service offers only one type of operation, which accepts only a zip code as an input. Thus, this service meets only half or the initial requirement. The third service accepts a city name as an input, thus makes it meets half of the requirement for the input. However, since the operation of the third service is to get a zip code from a city name, the output and operation requirements are not fulfilled. If the service description repository contains only these three service offers, the most relevant service would be the first service. The second service is ranked in the second place, and the third service will be omitted from the matching result.

The level of likeliness between requests and offers is determined by similarity values. When a similarity value is greater than a threshold value, then this offer and request are considered as "matched". The threshold level of similarity is adjustable. A higher threshold results in a potentially better match between the offer and the request. This may yield a better quality of results, but also reduces the number of potentially successful matches. Which threshold level should be chosen depends among other factors on the level of human involvement: If the results of the matchmaking process are used as recommendations or suggestions to human users, a lower threshold value might be advisable. If the matchmaking process results in direct invocation of matched services, a higher threshold is safer.

Service Request	**Service Offer (1)**
<types><schema...> <simpleType name=" **zipCode**" modelReference ="http://... #ZipCode"> </ simpleType > <simpleType name=" **cityName**" modelReference ="http://... geographydataset.owl#City"> </ simpleType > <complexType name=" **weatherForecast**" modelReference ="http://... #weather"> <sequence> <element name="**date**" /> <element name="**weatherDetail**" /> </sequence> </complexType> </schema></types>	<types><schema...> <simpleType name=" **ZipCode**" modelReference ="http://... #ZipCode"> </ simpleType > <simpleType name=" **PlaceName**" modelReference ="http://... geographydataset.owl#City"> </ simpleType > <complexType name="**Precipitation**" modelReference ="http://... #precipitation"> <sequence> <element name="**date**" /> <element name="**weatherDetail**" /> </sequence> </complexType> </schema></types>

<table>
<tr>
<td>

```
<message name=" WeatherByZipCodeRequest">
    <part name=" zipCode"/>
</message>
<message name=" WeatherByZipCodeResponse">
    <part name=" weatherDetail"/>
</message>
<message name="WeatherByCityNameRequest">
    <part name=" cityName"
</message>
<message name=" WeatherByCityNameResponse">
    <part name=" weatherDetail"/>
</message>

<portType name="WeatherForecast">
    <operation name="GetWeatherByZipCode">
        <input message=" WeatherByZipCodeRequest"/>
        <output message=" WeatherByZipCodeResponse"/>
    </operation>
    <operation name="GetWeatherByCityName">
        <input message=" WeatherByCityNameRequest"/>
        <output message="
        WeatherByCityNameResponse"/>
    </operation>
</portType>
```

</td>
<td>

```
<message name=" GetWeatherByZipCodeHttpGetIn">
    <part name=" zipCode"/>
</message>
<message name=" GetWeatherByZipCodeHttpGetOut">
    <part name=" weatherDetail"/>
</message>
<message name="GetWeatherByPlaceNameHttpGetIn">
    <part name=" cityName"
</message>
<message name=" GetWeatherByPlaceNameHttpGetOut">
    <part name=" weatherDetail"/>
</message>

<portType name="WeatherForecastHttpGet">
    <operation name="GetWeatherByZipCode">
        <input message=" WeatherByZipCodeRequest"/>
        <output message=" WeatherByZipCodeResponse"/>
    </operation>
    <operation name="GetWeatherByCityName">
        <input message=" WeatherByCityNameRequest"/>
        <output message=" WeatherByCityNameResponse"/>
    </operation>
</portType>
```

</td>
</tr>
<tr>
<th>Service Offer (2)</th>
<th>Service Offer (3)</th>
</tr>
<tr>
<td>

```
<types><schema...>
    <simpleType name=" ZipCode" modelReference
    ="http://... #ZipCode">
    </ simpleType >
    <complexType name="WeatherDescription"
    modelReference  ="http://... # weather">
        <sequence>
            <element name=" WeatherID" />
            <element name=" Description" />
        </sequence>
    </complexType>
</schema></types>

<message name=" GetCityWeatherByZIPHttpGetIn">
    <part name=" ZIP"/>
</message>
<message name=" GetCityWeatherByZIPHttpGetOut">
    <part name=" WeatherDescription"/>
</message>

<portType name=" WeatherHttpGet">
    <operation name=" GetCityWeatherByZIP">
        <input message="
        GetCityWeatherByZIPHttpPostIn"/>
        <output message="
        GetCityWeatherByZIPHttpPostOut"/>
    </operation>
</portType>
```

</td>
<td>

```
<types><schema...>
    <simpleType name=" ZipCode" modelReference
    ="http://... #ZipCode">
    </ simpleType >
    <complexType name="ArrayOfAnyType" >
        <sequence>
            <element name=" anyType" />
        </sequence>
    </complexType>
</schema></types>

<message name=" CityStateToZipCodeMatcherHttpGetIn">
    <part name=" City"/>
</message>
<message name=" CityStateToZipCodeMatcherHttpGetOut">
    <part name=" Body"/>
</message>

<portType name=" AddressLookupHttpPost">
    <operation name=" CityStateToZipCodeMatcher">
        <input message="
        CityStateToZipCodeMatcherHttpPostIn"/>
        <output message="
        CityStateToZipCodeMatcherHttpPostOut"/>
    </operation>
</portType>
```

</td>
</tr>
</table>

Table 6 Sample service request and offer for SAWSDL Matcher

Considering the example of the weather forecast service discovery, users might be interested in one specific city or any city they are currently staying. The request can be tailored to get results that are more specific. By using the current location information of the user instead of direct input from the request interface, MERCURY can make the result more specific and useful for the user.

Result Integrator

The results returned from service matchmakers, as presented in Klusch (2012), can be sorted according to the similarity score that is assigned to each web service. For example, the score of result from SAWSDL-iMatcher (Wei, Wang T., Wang J., and Bernstein, 2011) could be WSSAWSDL-MX = [WS1, WS2, WS3, WS4, WS5] = [0.9, 0.7, 0.6, 0.85, 0.93], where the elements in the array represent the similarity level of each service with respect to the request.

When there are more than one service matchmakers, the result integrator needs weighing parameters for each matchmaker. The weight parameter will be written and read from a configuration file. If it is not specified, the default value will be "1" for all matchmakers. All similarity scores of each web service from all service matchmakers will be accumulated and normalized, so that we can rearrange the ranking according to the final score.

Once again, the user context can play an important role here, in order to match the relevance of service recommendation to the need of users in each circumstance. Considering the following scenario, a user who resides in the USA would prefer to use the weather forecast service with a description "US Weather Service" rather than the one with a broader description like "Global Weather Forecast". Furthermore, a service with a description like "European Weather Service" can be neglected or moved to the lower rank since it does not match with the user context.

CONCLUSION

The MERCURY project aims to realize the seamless integration of devices and services with a user-friendly interface. In this paper, we mainly focused on the service discovery aspect exploiting personal awareness in order to improve the discovery results. We introduced a discovery module composed of a situation definition, a scenario definition and a scenario execution module. In order to address the requirements, we assumed that the service description contains at least a WSDL file. We also assumed that the application domain should be restricted, so that we can semantically analyse the request message properly.

The architectural overview described the main building blocks. The request converter reformats request and user context into compatible forms for service matchers. To improve the evaluation result, several service matchers may be used and the result integrator may merge the results. Finally, we reviewed related and existing work. At the moment, we are working on a proof-of-concept implementation.

FUTURE RESEARCH DIRECTIONS

Previously, we have proposed a prototype implementation of MERCURY. To accomplish the goal of "device and service smart integration", we need to integrate this module to MERCURY and exploit it along a practical use case. In addition, security concerns that may arise in particular with the personal awareness need to be addressed. We also consider the collaborative sharing, recommending and rating of "situation/scenario" templates among users. This can help to ease MERCURY accessibility for both personal and commercial use. Currently, we focus on the health insurance domain application for our next use case. We would like to deploy sensors and services such as blood sugar level sensor, training schedule and nutrition advisor. We have a foresight that the health condition tracking application would help the users and boost up service' values for the health insurance company.

ACKNOWLEDGEMENT

This research is accomplished under the framework of the Mercury project and is supported by IBM Deutschland Research & Development GmbH, Foundation "Gran Mariscal de Ayacucho" and DAAD.

REFERENCES

Blake, M. B., Bleul, S., Weise, T., Bansal, A., & Kona, S. (2009). *Web Services Challenge*, Retrieved February 21, 2013, from http://ws-challenge.georgetown.edu/wsc09.

Blake, M. B., Cabral, L., König-Ries, B., Küster, U. & Martin, D. (2012). *Semantic Web Services : Advancement through Evaluation.* Berlin: Springer.

Bo, C., Xiuquan, Q., Budan, W., Xiaokun, W., Ruisheng, S., & Junliang, C. (2012). *RESTful Web Service Mashup Based Coal Mine Safety Monitoring and Control Automation with Wireless Sensor Network.* Paper presented at IEEE 19th International Conference on Web Services (ICWS), pp. 620-622. IEEE.

Breslin, J. G., Passant A., & Decker S. (2009). *The Social Semantic Web (1st ed.).* Springer.

Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). *Web Services Description Language (WSDL).* Retrieved February 21, 2013, from http://www.w3.org/TR/wsdl

Cloud Business Apps Integration – CloudWork. (2013). Retrieved February 21, 2013, from https://cloudwork.com/

Cosm - Internet of Things Platform Connecting Devices and Apps for Real-Time Control and Data Storage. (2012). Retrieved February 21, 2013, from https://cosm.com

Ding, Z., Lei, D., Yan, J., Bin, Z., & Lun, A. (2010). *A Web Service Discovery Method Based on Tag*. Paper presented at International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp.404-408. IEEE Computer Society.

Dey, A. K., Sohn, T., Streng, S., & Kodama, J. (2006). iCAP: Interactive prototyping of context-aware applications. In proceedings of *Pervasive Computing, Vol. 3968* (pp.254-271). Berlin, Heidelberg : Springer.

Evrythng. (2012). Retrieved February 21, 2013, from http://www.evrythng.com

Gawinecki, M. (2009). *WSColab: Structured Collaborative Tagging for Web Service Matchmaking*. Retrieved February 21, 2013, from http://www.ibspan.waw.pl/~gawinec/wss/wscolab.html

Gawinecki, M., Cabri, G., Paprzycki, M., & Ganzha, M. (2010). WSColab: Structured Collaborative Tagging for Web Service Matchmaking. In Proceedings of *the 6th International Conference on Web Information Systems and Technologies (WEBIST 2010), Vol. 1* (pp.70-77). Valencia, Spain : INSTICC Press.

Gawinecki, M., Cabri, G., Paprzycki, M., & Ganzha, M. (2012). Evaluation of Structured Collaborative Tagging for Web Service Matchmaking. *Semantic Web Services Advancement through Evaluation*, pp. 173-189. Berlin, Heidelberg : Springer.

Gray, A. J. G., Castro, R. G., Kyzirakos, K., Karpathiotakis, M., Calbimonte, J. P., Page, K., Sadler, J., Frazer, A., Galpin, I., Fernandes, A. A. A., Paton, N. W., Corcho, O., Koubarakis, M., Roure, D. D., Martinez, K., & Perez, A. G. (2011). A semantically enabled service architecture for mashups over streaming and stored data. In ESWC'11, *the 8th extended semantic web conference on the semantic web: research and applications, Vol. part II* (pp. 300-314). Berlin, Heidelberg : Springer.

Grosky, W.I., Kansal, A., Nath, S., Liu, J., & Zhao, F. (2007). SenseWeb: An Infrastructure for Shared Sensing. *IEEE Multimedia, 14*(4), 8-13.

Guinard, D. (2009). *Towards the web of things: Web mashups for embedded devices*. Paper presented at 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web MEM 2009, Madrid, Spain.

Guo, B., Zhang, D., & Imai M. (2011). Toward a Cooperative Programming Framework for Context-Aware Applications. *Personal and Ubiquitous Computing, 15*(3), 221-233.

Harth, A., & Maynard D. (2012). *Semantic Web Challenge*. Retrieved February 21, 2013, from http://challenge.semanticweb.org

Hobold, G.C., & Siqueira, F. (2012). *Discovery of Semantic Web Services Compositions Based on SAWSDL Annotations.* Paper presented at IEEE 19th International Conference on Web Services (ICWS), pp.280-287.

IBM WebSphere Portal.(1994, 2013). Retrieved February 21, 2013, from http://www-01.ibm.com/software/websphere/portal/

IFTTT - If this then that. (2013). Retrieved February 21, 2013, from https://ifttt.com

Klusch, M., Kapahnke, P., & Zinnikus, I. (2009). SAWSDL-MX2: A Machine-Learning Approach for Integrating Semantic Web Service Matchmaking Variants. *Web Services, IEEE International Conference on ICWS 2009,* pp. 335-342.

Klusch, M. (2012). *Semantic Service Selection (S3) contest*. Retrieved February 21, 2013, from http://www-ags.dfki.uni-sb.de/~klusch/s3/index.html

Koenig-Ries, B., Opasjumruskit, K., Nauerz, A., & Welsch, M. (2012). *MERCURY : User Centric Device & Service Processing*. Paper presented at Mensch & Computer Workshopband (MKWI), Braunschweig, Germany, pp. 103-106.

Kovatsch, M., Lanter, M., & Duquennoy, S. (2012). Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications. In Proceedings of *the 3rd International Conference on the Internet of Things (IoT)*, pp. 135- 142.

Maleshkova, M., Kopeck, J., & Pedrinaci, C. (2009). Adapting SAWSDL for Semantic Annotations of RESTful Services. In Proceedings of *the Confederated International Workshops and Posters on "On the Move to Meaningful Internet Systems"*, pp. 917-926. Berlin, Heidelberg : Springer.

Martin, D., Paolucci M., Mcilraith, S., Burstein, M., Mcdermott, D., Mcguinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., & Sycara, K. (2004). Bringing Semantics to Web Services: The OWL-S Approach. In Proceedings *of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, pp. 26-42. Heidelberg : Springer.

Martin, D., Paolucci, M., & Wagner, M. (2007). Bringing semantic annotations to web services: OWL-S from the SAWSDL perspective. In Proceedings *of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, pp. 340-352. Berlin, Heidelberg : Springer.

Mattern, F., & Floerkemeier, C. (2010). From the Internet of Computers to the Internet of Things. Lecture Notes in *Computer Science Vol. 6462*, Berlin, Heidelberg : Springer.

On{X} – automate your life. (2012). Retrieved February 21, 2013, from https://www.onx.ms

Ngan, L. D., Kirchberg, M., & Kanagasabai, R. (2010). *Review of Semantic Web Service Discovery Methods*. Paper presented at IEEE 6th World Congress on Services (SERVICES-1), pp. 176-177.

Opasjumruskit, K., Exposito, J., Koenig-Ries, B., Nauerz, A., & Welsch, M. (2012, September) *MERCURY: User Centric Device and Service Processing – Demo paper*. Paper presented at 19th Intl. workshop on Personalization and Recommendation on the Web and Beyond, Mensch & Computer 2012, Konstanz, Germany.

Paraimpu - The Web of Things is more than Things in the Web. (2012). Retrieved February 21, 2013, from http://paraimpu.crs4.it/

Phuoc, D. L., & Hauswirth, M. (2009). Linked open data in sensor data mashups. In Proceedings of *the 2nd International Workshop on Semantic Sensor Networks (SSN09) in conjunction with ISWC 2009, Vol. 522* (pp. 1-16).

Roman, D., Keller, U., Lausen, H., Bruijn, J.D., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., & Fensel, D. (2005). Web Service Modeling Ontology. *Applied Ontology 1*(1), 77-106.

Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative filtering recommender systems. *The adaptive web, Vol. 4321*, pp. 291-324. Berlin, Heidelberg : Springer.

Soylu, A., Causmaecker, P., & Desmet, P. (2009). Context and Adaptivity in Pervasive Computing Environments: Links with Software Engineering and Ontological Engineering. *Journal of Software, 4*(9), 992-1013 .

Talantikite, H. N., Aissani, D., & Boudjlida, N. (2009). Semantic annotations for web services discovery and composition. *Computer Standards & Interfaces, 31* (6), 1108-1117.

Tran, V.X., Puntheeranurak, S., & Tsuji, H. (2009). *A new service matching definition and algorithm with SAWSDL*. Paper presented at 3rd IEEE International Conference on Digital Ecosystems and Technologies (DEST '09), pp. 371-376.

Wei, D., Wang, T., Wang, J., & Bernstein, A. (2011). SAWSDL-iMatcher: A customizable and effective Semantic Web Service matchmaker. *Web Semantics: Science, Services and Agents on the World Wide Web, 9*(4), 402-417.

KEY TERMS & DEFINITIONS

User centric: A stage of users being more powerful and able to take control over the devices or systems, in contrast to the traditional device centric technology. Thus, the users can exploit more from devices and systems regardless of operating systems, hardware or locations.

Service discovery: An ability of automatically detecting physical devices and services, which are available on the computer network. Users are able to query for a service provider, then further access and exploit the required service.

Adaptive computing : A computing system that can adjust to the environmental factors during the runtime. An adaptive system may change the behavior or functionality according to individual users or situations.

Context awareness: A property of a device, especially mobile devices, to sense the surrounding information, such as, location, temperature, etc. This term is coined from ubiquitous computing, since it is the bridge connecting between computer systems and physical environment.

Semantic Service Description: A service description, normally provided by the service provider, describes the functionality and instructs how to use the service. Not only consumable by machines, but

human can also get benefits from the description. When the semantic service description is extended with semantic annotations, textual information will be enhanced with meanings derived from ontologies.

Service Recommendation: In the context of MERCURY, once a service has been registered to the system, though by the other users, a user can receive a suggestion to add a service that is relevant to the contexts, preferences and expertises of the user.

Collaborative description: An effort to share service descriptions among users in the same system. Therefore, the services, which may or may not contain descriptions from service providers, can be discovered and recommended to the other users, with a minimum cold-starting problem.