A System for Suggestion and Execution of Semantically Annotated Actions based on Service Composition

Milos Jovanovik, Petar Ristoski, Dimitar Trajanov

Faculty of Computer Science and Engineering, Ss. Cyril and Methodius in Skopje, Republic of Macedonia

milos.jovanovik@finki.ukim.mk, petar.ristoski88@gmail.com, dimitar.trajanov@finki.ukim.mk

Abstract. With the growing popularity of the service oriented architecture concept, many enterprises have large amounts of granular web services which they use as part of their internal business processes. However, these services can also be used for ad-hoc actions, which are not predefined and can be more complex and composite. Here, the classic approach of creating a business process by manual composition of web services, a task which is time consuming, is not applicable. By introducing the semantic web technologies in the domain of this problem, we can automate some of the processes included in the develop-and-consume flow of web services. In this paper, we present a solution for suggestion and invocation of actions, based on the user data and context. Whenever the user works with given resources, the system offers him a list of appropriate actions, preexisting or ad-hoc, which can be invoked automatically.

Keywords: Semantic web services, automatic composition, semantic web technologies, service oriented architecture.

1 Introduction

The growing trend in software architecture design is to build platform-independent software components, such as web services, which will then be available in a distributed environment. Many businesses and enterprises are tending to transform their information systems into linked services, or repeatable business tasks which can be accessed over the network. This leads to the point where they have a large amount of services which they use as part of predefined business processes. However, they face the problem of connecting these services in an ad-hoc manner.

The information an employee works with every day, can be obtained from different sources – local documents, documents from enterprise systems or other departments, emails, memos, etc. Depending on the information, the employee usually takes one or more actions, such as adding a task from an email into a To-Do list, uploading attachments to another company subsystem for further action or analysis, or sending the attachments to the printer.

Additionally, with the increasing number of cloud services with specialized functionalities in the last years, the common Internet user comes across the need to routinely perform manual actions to interchange data among various cloud services – email, social networks, online collaboration systems, documents in the cloud, etc. – in order to achieve more complex and composite actions. These actions always require a certain amount of dedicated time from the user, who has to manually change the context in which he or she works, in order to take the appropriate actions and transfer data from one system to another.

In this paper we present a way of using the technologies of the Semantic Web [1], to automate the processes included in the develop-and-consume flow of web services. The automatic discovery, automatic composition, and automatic invocation of web services provide a solution for easier, faster and ad-hoc use of specialized enterprise services for an employee in the company, and of public services for the common Internet user.

The paper is structured as follows: In Section 2 we provide an overview of existing related solutions and approaches. In Section 3 we give a detailed explanation of the system architecture and its components. In Section 4 we describe the algorithm for detection and selection of the most suitable action for returning the requested output from the set of provided inputs. In Section 5 we discuss the advantages and applications of the system. We conclude in Section 6 with a short summary and an outlook on future work.

2 Related Work

As the semantic web technologies proved their usability in a large number of IT systems [2], [3], and as most of the applications and systems are now being built upon the Service Oriented Architecture (SOA) model [4], many solutions combining the two fields have been developed. These solutions apply semantic web technologies into SOA systems, in order to automate various complex processes within them [5], [6].

There are many tools and solutions for designing and running standard BPEL processes, such as Oracle Fusion Middleware[1] and IBM Websphere[2] [7]. However, they usually don't provide the ability to describe and characterize the services with semantics. Without information about the service capabilities and behavior, it is hard to compose collaborative business processes.

One of the solutions for this problem is the OntoMat-Service [8], a framework for discovery, composition and invocation of semantic web services. OntoMat-Service does not aim at intelligent and completely automatic web service discovery, composition and invocation. Rather, it provides an interface, the OntoMat-Service-Browser, which supports the intelligence of the user and guides him or her in the process of adding semantic information, in a way that only a few logically valid paths remain to be chosen.

[1] http://www.oracle.com/technology/products/middleware/index.html
[2] http://www.ibm.com/software/websphere/

The system described in [9] can deal with preexisting services of standard enterprise systems in a semantically enriched environment. By transforming the classic web services into semantic web services, the services are prepared to be invoked within a prebuilt business process. The system described in [10] presents a web service description framework, which is layered on top of the WSDL standard, and provides semantic annotations for web services. It allows ad-hoc invocation of a service, without prior knowledge of the API. However, this solution does not support the ability of creating a composition of atomic semantic web services.

The authors in [11] propose a planning technique for automated composition of web services described in OWL-S process models, which can be translated into executable processes, like BPEL programs. The system focuses on the automatic composition of services, disregarding the user's context and provided inputs to suggest the most reliable and relevant composition.

Another approach [12] describes an interface-matching automatic composition technique that aims to generate complex web services automatically by capturing user's expected outcomes when a set of inputs are provided; the result is a sequence of services whose combined execution achieves the user goals. However, the system always requests the user's desired output, which means that the system is unable to suggest new actions. Additionally, it is not guaranteed that the system would always choose the most reliable compositions of services, as the compositions are built based only on two factors: the execution time and the similarity value between the services in the composition, expecting only one user input.

Similar approaches have been further studied in [13], [14] and [15]. However, none of the related systems fully automate the workflow of discovery, ranking and invocation of web services and web service compositions, but they only automate a certain part of it. In our solution, we fully automate the workflow of web service and web service composition invocation, which includes automatic fetching of possible actions for a given context, automatic ranking and composition, and automatic invocation.

3 Solution Description

Our approach is based on web service invocation. We refer to the invocation as *taking an action*. As an *action* we consider a single RESTful service, a single SOAP web service, or a composition of more than one SOAP web services. The system tries to discover all of the possible actions that can be taken over the given resources in a given context, and provides the user with a list of available actions to execute. The user can then quickly execute complex actions by a single click. These actions can be discovered in an ad-hoc manner, i.e. they do not have to be predefined and premodeled.

The solution is developed in the Java programming language, using the Play MVC framework[3]. The system architecture, shown in Fig. 1, consists of several components.

---

[3] http://www.playframework.org/

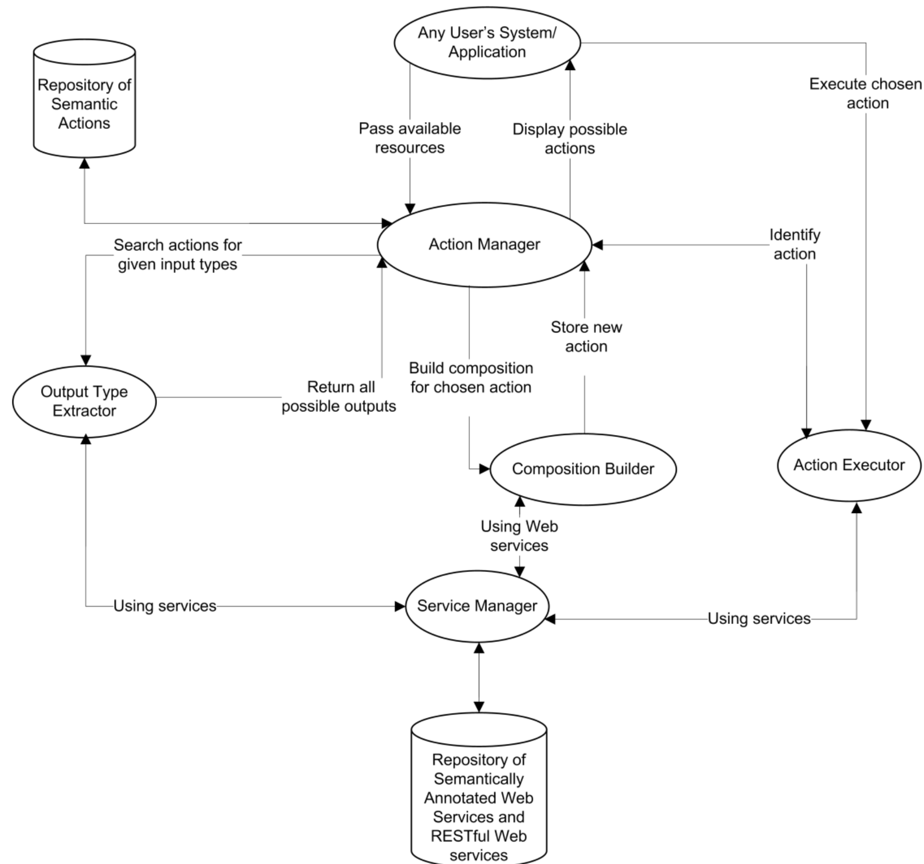

Fig. 1. System Architecture

The *repository of semantically annotated web services and RESTful web services* (SAWSRWS) holds information about all of the semantically annotated web services. There is no restriction on the technology used to develop the web services. After a web service is developed, in order for it to be uploaded onto the repository it has to be semantically annotated. The system provides a simple form for annotation and saving the information for the new web services into the repository. For SOAP web services, the semantically annotated WSDL file for the service is stored. The SOAP web services are annotated using the Semantic Annotations for WSDL and XML Schema (SAWSDL) framework[4]. For RESTful web services, we store an XML file with details about the service, such as the base-URL, the method type, the names of the input parameters and the output parameter, along with their semantic annotations. This information is stored within an XML file in the repository.

[4] http://www.w3.org/TR/sawsdl/

The *service manager* is responsible for handling the requests to the SAWSRWS repository: adding, removing, updating and loading services. On system startup, the manager is indexing the services, and this index is updated only when a service is added or removed from the SAWSRWS repository. With this index, the number of accesses to the SAWSRWS repository is decreased and the system performance is improved. Newly created actions are stored in the *repository of semantic action* in three strictly defined storage forms. The first storage form includes a unique ID for the action, the inputs that are needed to invoke the action, and the output of the action. The second storage form is an upgrade of the first storage form, which includes a list of all the web services that compose the action, which are described with their name, the function, and the inputs and output of the function. The third storage form is used only for RESTful web services, which cannot be part of a composition in our system. This storage form includes the base-URL of the RESTful web service, the input parameters, the output parameter and the method type.

The *action manager* is intended to improve the performance of the system. On system startup, if the repository is not empty, the action manager creates an index of the actions. The action manager is handling the requests from the user applications and systems. When a request arrives, the resource types are aligned as inputs. The action manager is iterating the index to find if previously created actions for these inputs exist. If not, the action manager sends a request with the list of inputs to the *output type extractor*, and receives a list of all possible outputs that can be obtained from the available services. From the list of outputs, the action manager creates actions in the first storage form, for SOAP web services, and creates actions in the third storage form for RESTful web services. Then it stores them in the semantic action repository, updates the action index and returns the list of actions, in XML form, to the user application or system. When the user wants to invoke an action, a request to the *action executor* is sent, which identifies the action in the action manager, based on the action unique ID. The action manager checks in the index of actions for the storage form of the action with the given ID. If the action is in the second or third storage form, the actions' details are sent to the action executor. If the action is in first storage form, the action manager sends the action details to the *composition builder*, and receives a composition of functions from the web services with their name, list of inputs and the output. Then the action storage form is upgraded to the second storage form, and the action details are sent to the action executor.

The *output type extractor* receives the list of inputs from the action manager and iterates the index of services in the service manager, in order to find all of the possible outputs from the services for the given list of inputs. In the list of outputs we add only the outputs of the service functions which can be invoked with the given list of inputs, or a subset of the list of inputs. When a new output is detected, it is added both to the list of outputs and to the list of inputs. Then the extractor iterates the index of services again, with the new list of inputs. When there are no more new outputs, the iteration stops. Then the extractor sends the list of detected outputs to the action manager.

The *composition builder* receives a list of inputs and one output from the action executor. The composition builder uses an intelligent algorithm, described further, for building an optimal composition of semantic web services, in order to provide the

needed output for the list of given inputs. The RESTful web services are not included in compositions.

The *action executor* receives requests from the user applications and systems. The request contains the action ID and a list of values, which represent the input values for the action. The action executor identifies the action in the action manager, and receives a RESTful web service, a single function from a SOAP web service, or a composition of functions from SOAP web services, ordered for invocation. In the former two cases, the invocation is done in a single step. But for the latter case, the action executor invokes the first SOAP web service function, for which every input value is provided by the user. If the function is successfully executed, the result value is added in the initial list of input values, and the values used for invocation are removed from the list. The same steps are repeated for the rest of the functions. When the last function is executed, if the function has an output, it is displayed to the user; otherwise, a message for successful invocation is displayed to the user. If any function fails to execute, the algorithm stops, and an error message is displayed to the user. For the invocation of the actions we use the Apache Axis2 engine[5].

4 Service Composition

The composition builder uses a specially created algorithm for building an optimal composition of semantic web services, in order to provide the needed output for a list of given inputs. The algorithm works with a set of inputs and an output parameter, provided to the composition builder by the action manager. The algorithm tries to identify if the service repository contains a single service which returns an output of the same semantic type, as the requested output value. If there are one or more such services, it checks to see if their input parameters match the inputs provided to the composition builder.

If o_r is the semantic type of the requested output parameter, and o_i is the semantic type of the output from the ith web service from the repository, what the algorithm tries to find are services for which

$$o_r = o_i \tag{1}$$

is true. These semantic web services become candidate web services for providing the requested output.

For each of the semantic web services which satisfy the equation (1), the algorithm has to compare the input types set $I_r = \{i_{r1}, i_{r2}, ..., i_{rn}\}$, provided to the composition builder, and the set of input types of the ith web service, $I_i = \{i_{i1}, i_{i2}, ..., i_{im}\}$. If the two sets satisfy that

$$|I_i| < |I_r| \tag{2}$$

[5] http://axis.apache.org/axis2/java/core

the algorithm eliminates the ith semantic web service from the list of potential candidate web services, because the number of input parameters of the services is less than the number of input parameters provided to the composition builder. This way, the potential lack of precision in the output, caused by lesser constraints, is eliminated.

If the input sets satisfy that

$$I_i = I_r \quad (3)$$

it means that the number and the semantic types of the inputs provided to the composition builder match the number and the semantic types of the ith web service. The algorithm assigns this semantic web service with a *fitting coefficient*:

$$F = 1$$

The fitting coefficient – F, represents the suitability of a given semantic web service, or a composition of semantic web services, to provide the requested output.

When the ith web service satisfies (3), the service is considered to be the most suitable – the requested output can be returned in just one step. Therefore its fitting coefficient is equal to the highest value. When the algorithm discovers at least one semantic web service with $F = 1$, the discovery of candidate web services ends.

If the algorithm does not find a suitable semantic web service for the received request, i.e. does not find a service which satisfies (1), the algorithm ends without success and does not return a suitable semantic web service or a composition of semantic web services.

If the sets of input satisfy that

$$|I_i| \geq |I_r| \quad (4)$$

but the types of all of the input parameters of the ith semantic web service do not match the types of the input parameters provided to the composition builder, the fitting coefficient of the ith semantic web service is calculated as

$$F_i = \frac{y_{fi}}{y_i} \quad (5)$$

where $y_{fi} = |I_{fi}|$ is the number of parameters from the ith semantic web service which have a matching semantic type with the input parameters from I_r, $I_{fi} \subseteq I_i$, and $y_i = |I_i|$.

In this case, the algorithm continues to search for the other input parameters which do not belong to I_{fi}. The algorithm starts again, but now the requested output o_r is the input parameter which does not belong to I_{fi}. If we have more than one such parameter, this secondary search is performed for each of them. This way, we search for outputs from other services which can be used as inputs for the discovered service.

If the services discovered in the secondary search have the same types of input parameters as the inputs provided to the composition builder, they can be invoked and their outputs can be used as inputs for the semantic web service discovered in the first iteration. If they too have input parameters with types which do not match those provided to the composition builder, the algorithm performs a tertiary search for services which can provide them. These iterations last until the algorithm does not come to the state in which all of the discovered semantic web services have the same types of input parameters as the input parameters provided to the composition builder and as the outputs provided from other services in the composition, or the state in which a suitable service or composition cannot be discovered.

By creating a composition of semantic web services in this manner, we raise the cost for getting the required output. Depending on the number of services and levels in the composition, the time necessary to get the output from the list of given inputs increases. Additionally, as the composition grows larger, so does the possibility of an error occurring during a call to a web service from the composition. Therefore, we must somehow take this into account in our calculations for the fitting coefficient.

We add a *coefficient for fitness degradation*:

$$K_i = \sum_{j=1}^{p} (k_1^{-1} + y_j k_2^{-1}) \qquad (6)$$

where p is the total number of services in the composition, without the initially discovered service, y_j is the number of input parameters from the jth service, and k_1 and k_2 are *factors for fitness degradation*. k_1 is a factor of influence of the number of services from the composition. k_2 is a factor of the influence of the number of parameters of services from the composition. Generally, the values of these factors should always be $k_1 < k_2$, because the number of services has a bigger impact on the total call time of the composition, compared to the number of parameters of the services. The default values for the factors are chosen to be $k_1 = 10$ and $k_2 = 100$, and can be modified within the composition builder.

From (6) we can see that the algorithm does not take into account the level of composition at which the jth service is positioned. This is because the calls to web services from the same level are performed sequentially, just as the calls to web services from different levels. Therefore, the cost for getting the requested output depends only on the number of services in the composition, and not their level distribution.

From (6) we can also see that the coefficient depends on the number of parameters used for each of the services, disregarding whether they are provided to the composition builder, or returned from another service. This is because the number of parameters represents the amount of data which has to be transferred for the calls to the services, so the nature of the parameters is irrelevant.

We add the coefficient for fitness degradation to (5):

$$F_i = \frac{y_{fi}}{y_i} - K$$

$$F_i = \frac{y_{fi}}{y_i} - \sum_{j=1}^{p} (k_1^{-1} + y_j k_2^{-1}) \quad (7)$$

The algorithm uses (7) to calculate the fitness of all candidate semantic web services which satisfy (4). The fitting coefficient is larger when a candidate service uses more of the input parameters provided to the composition builder. The coefficient drops with the number of additional services and the number of their input parameters.

Once the algorithm calculates F for each of the candidate web services, they are ranked and the atomic service or a composition of services with the highest value of F is selected as most suitable for providing the requested output.

5 Advantages and System Usability

The flexible architecture of the solution allows it to be used from within various systems. In order for it to be prepared for use in a new domain, it requires semantically annotated services from the domain, which can be taken from different enterprise systems and cloud infrastructures. After this step is completed, the users can receive a list of possible actions for any resources and data they are working with, within their own environment. These ad-hoc actions can then be executed by a single click, which is time-saving; an advantage towards which all modern tools aim. Additionally, because of its modularity, the solution can be easily updated and extended.

5.1 Use-Case

In this scenario, the user application works with geographic data, and it uses several web services which have the functions given in Table 1.

In this use-case, a user uses an application which works with the web services from Table 1, and has a name of a certain municipality as the only useful information in the context of the working environment, e.g. an email message. In this case, the application has only one service which can be invoked for the context of the user – WSF6 from Table 1 – so in a standard SOA architecture with service discovery the system will only offer this action to the user.

Table 1. List of web service functions from the use-case example.

WSF #	*Web Service Function*
1	DialingCode *getDialingCode* (Country country, City city);
2	DialingCode *getDialingCode* (Continent continent, Country country, City city, Municipality municipality);
3	Continent *getContinent* (Country country);
4	Country *getCountry* (Municipality municipality, City city);
5	Country *getCountry* (City city);
6	City *getCity* (Municipality municipality);
7	City *getCity* (City city);

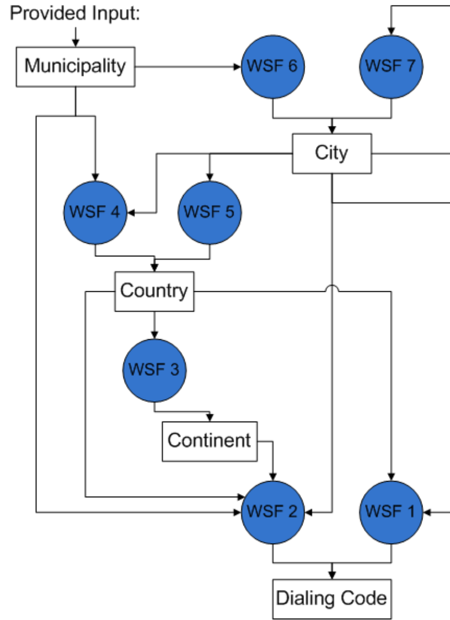

Fig. 2. Possible action for the given user input, i.e. the name of the municipality.

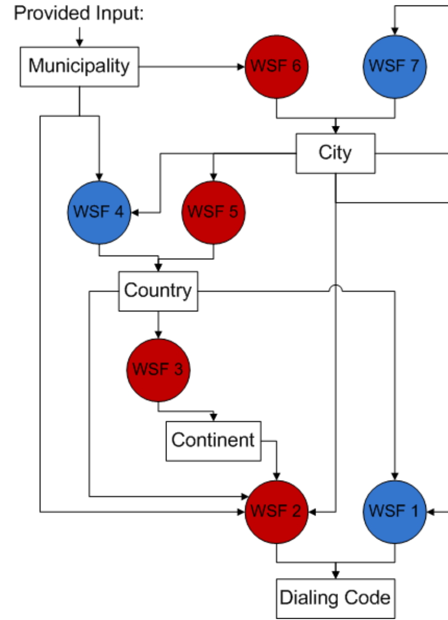

Fig. 3. The most suitable semantic web service composition for returning a dialing code, based on the name of the municipality.

However, if the application is connected with the system presented in this paper, the number of possible actions will grow. The system, besides WSF6 as an atomic web service, will automatically detect the possible semantic web service compositions, as shown on Fig. 2. This means that four different actions: *getCity*, *getCountry*, *getContinent* and *getDialingCode* can be executed over the information extracted from the context of the user, i.e. the name of the municipality, either as atomic web services or as web service compositions. When an action represents a composition of SOAP web services, its name is derived from the name of the last web service function in the composition. When the action consists of a single SOAP or RESTful web service, it has the same name as the atomic web service.

These actions are built from the most optimal and most reliable web services or compositions of web services, according to the algorithm from Section 4. One of these actions is the action which can return the dialing code, based only on the municipality name. This action can be derived from four different web service compositions, shown in Table 2, all of which end with either WSF1 or WSF2. Therefore, the name of this action is *getDialingCode*.

For each of the compositions which return the dialing code, we calculate the fitting coefficient F (for $k_1 = 10$ and $k_2 = 100$), using (7), and the results are shown in Table 2. The most suitable composition, the one with the highest value of F, is chosen for representing the action of deriving the dialing code based on the municipality name. In this scenario, it is the first composition.

Table 2. List of possible compositions for providing the dialing code based on the municipality name.

Action	*Fitting Coefficient*
WSF6 → WSF5 → WSF3 → WSF2	$F = 0.19$
WSF6 → WSF4 → WSF3 → WSF2	$F = 0.18$
WSF6 → WSF4 → WSF1	$F = -0.17$
WSF6 → WSF5 → WSF1	$F = -0.18$

The user can choose to execute any of the actions from Fig. 2, just by a single click. The list contains all of the possible actions for the given input, by creating compositions of web services. Thus, introducing new actions, which were previously not part of the user system, or the user was unaware that they existed, is done automatically. Even if the user was aware that these actions were available, he or she would have had to pre-connect the services manually into a business process, which takes much longer and is not an easy task to do. This is essential in systems where the services are constantly changing, and new services are added regularly.

5.2 Application

The solution has been implemented as part of Semantic Sky, a platform for cloud service integration [16]. Semantic Sky enables connectivity and integration of different cloud services and of local data placed on the users machines, in order to create a simple flow of information from one infrastructure to another. It is able to automatically discover the context in which the users are working, and based on it and by using the solution described in this paper, provide them with a list of actions which can be executed over their data. In this way, the users can completely focus on their tasks in their work environment, and get relevant information and executable actions in their current context. By automating the discovery and execution of relevant tasks, the system improves the productivity, information exchange and efficiency of the users.

6 Conclusion and Future Work

This paper presents a solution for automatic discovery and invocation of atomic web services and web service compositions, by employing semantic web technologies. The solution provides a list of all possible actions which exist within the user system or in distributed environments and which can be executed over the data and information the user is currently working with. This approach offers the user a broader perspective and can introduce action for which he or she was previously unaware.

Additionally, the solution offers actions in an ad-hoc manner; the service compositions are created on-the-fly, overriding the need for pre-connecting the services into fixed compositions, i.e. creating pre-built business processes. This is essential in systems where the services are continually changing, and new services are added regularly. In such dynamic environments, the fast and automatic detection of new possible actions is of high importance.

Currently, the solution does not support building compositions of RESTful web services or combining RESTful web services with SOAP web services in a same composition. This is because we use primitive data types for semantic annotation of the inputs and the output of the services, and most of the RESTful services return a more complex value. This can be solved by adding more complex classes for annotation into the ontologies. These drawbacks will be our main focus in the future development of the solution.

References

1. Berners-Lee, T., Hendler. J. Lassila, O.: The Semantic Web. Scientific American, (2001)
2. Hitzler, P., Krotzsch, M. and Rudolph, S.: Foundations of Semantic Web Technologies. Chapman and Hall/CRC (2011)
3. Sugumaran, V., Gulla, J. A.: Applied Semantic Web Technologies. Auerbach Pub (2012)
4. He, H.: What is Service-Oriented Architecture?, O'Reilly XML.com, (2003)
5. Hu, Y., Yang, Q., Sun, X. and Wei, P.: Applying Semantic Web Services to Enterprise Web. International Journal of Manufacturing Research, vol. 7 (1), 1-8. DOI:10.1504/IJMR.2012.045240 (2011)
6. Wang, D., Wu, H., Yang, X., Guo, W. and Cui, W.: Study on Automatic Composition of Semantic Geospatial Web Service. In: IFIP Advances in Information and Communication Technology, vol. 369/2012, 484-495, DOI:10.1007/978-3-642-27278-3_50 (2012)
7. Kloppmann, M. Business process choreography in WebSphere: Combining the power of BPEL and J2EE. IBM Systems Journal, vol. 43 (2), 270-296 (2004)
8. Agarwal, S., Handschuh, S. and Staab, S.: Annotation, Composition and Invocation of Semantic Web Services. Journal of Web Semantics, vol. 2 (1), 1-24 (2004).
9. Martinek, P., Tothfalussy, B. and Szikora, B.: Execution of Semantic Services in Enterprise Application Integration. In: World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 128-134 (2008)
10. Eberhart, A.: Ad-hoc Invocation of Semantic Web Services. In: IEEE International Conference on Web Services (Las Vegas, USA, 2003), IEEE Computer Society, Washington, DC, USA, 116-123. DOI:10.1109/ICWS.2004.18 (2004)
11. Traverso, P., Pistore M.: Automated Composition of Semantic Web Services into Executable Processes. In: The third International Semantics Web Conference (ISWC-04), Springer, vol. 3298, pp. 380-394, Hiroshima, Japan (2004)
12. Zhang, R., Arpinar, B., Aleman-Meza, B.: Automatic Composition of Semantic Web Services. In: ICWS 2003: pp. 38-41 (2003)
13. He, T., Miao, H., Li, L.: A Web Service Composition Method Based on Interface Matching. In: The eight IEEE/ACIS International Conference on Computer and Information Science (2009)
14. Talantikite, H., Aissani, D., Boudjilda, N.: Semantic Annotations for Web Services Discovery and Composition. Computer Standards & Interfaces archive Volume 31 Issue 6, November, pp. 1108-1117 (2009)
15. Hamadi, R., Benatallah, B.: A Petri Net-Based Model for Web Service Composition. In: The 14th Australasian database conference, pp. 191-200, Adelaide, Australia (2003)
16. Trajanov, D., Stojanov, R., Jovanovik, M., Zdraveski, V., Ristoski, P., Georgiev, M., Filiposka, S.: Semantic Sky: A Platform for Cloud Service Integration based on Semantic Web Technologies. In: Proceedings of the 8th International Conference on Semantic Systems (I-SEMANTICS '12). ACM, New York, NY, USA, 109-116 (2012)