An Algorithm for Automatic Service Composition

Eduardo Silva, Luís Ferreira Pires and Marten van Sinderen

Centre for Telematics and Information Technology, University of Twente
Enschede, the Neterlands

Abstract. Telecommunication companies are struggling to provide their users with value-added services. These services are expected to be context-aware, attentive and personalized. Since it is not economically feasible to build services separately by hand for each individual user, service providers are searching for alternatives to automate service creation. The IST-SPICE project aims at developing a platform for the development and deployment of innovative value-added services. In this paper we introduce our algorithm to cope with the task of automatic composition of services. The algorithm considers that every available service is semantically annotated. Based on a user/developer service request a matching service is composed in terms of component services. The composition follows a semantic graph-based approach, on which atomic services are iteratively composed based on services' functional and non-functional properties.

1 Introduction

Advances in mobile communications and devices triggered a multitude of new and innovative services and business areas. For example, value-added services are being proposed by telecommunication companies and service providers, aiming to provide personalized, context-aware and attentive services to end-users. The IST-SPICE (Service Platform for Innovative Communication Environment) project [1] aims at developing a platform to be used by end-users and application developers for the development and deployment of innovative services. SPICE services are composed based on a collection of components, whose services can be published and used in service compositions by end-users and application developers. This is made possible by applying Web Services technology [2] and the Service-Oriented Architecture (SOA) principles [3].

Since it is not economically feasible to build services separately by hand for each individual user, and furthermore, it is hard to predict at design time what personalised services the user may wish, a lot of attention has been given lately to the (semi-) automatic composition of services [9, 10]. Automatic service composition starts with a service request describing the service desired by an end-user or service developer. If there is no service already available that matches the request, a composition of other available services that match the request is constructed. These available services are denoted here as atomic services. In the SPICE project these services are specified in a language called SPATEL (SPice Advanced service description language for TELe-communication services) [4]. This language allows the specification of services in a platform independent manner, including annotations concerning semantic and non-functional properties. Such annotations are imperative in the context of automatic service composition.

One of the activities of the SPICE project is the development of an Automatic Composition Engine (ACE), which should support end-users and application designers on the development of service compositions. The ACE is expected to receive either service requests from end-users in natural language, or from the application designers in some well-defined notation, and deliver a service composition or a list of alternative compositions, respectively. This paper discusses our ideas concerning the automatic service composition algorithm. Our approach relies on the use of semantic annotations on the atomic services, and on service requests, to perform the service discovery, matching and composition. We define our approach as a semantic graph-based automatic composition, where discovered services are represented in a graph, which is used to optimize the search of component services and their composition.

The paper is further organized as follows: section 2 provides some motivation for our work, including a motivating example of application of automatic service composition; section 3 gives an overview of the SPICE Automatic Composition Engine; section 4 presents our initial approach towards the algorithm for automatic service composition; section 5 compares our approach with some related work; and section 6 presents our conclusions and depicts directions for future work.

2 Automatic Service Composition

Automatic service composition aims at automatically composing services that satisfy a given service request from an end-user or service developer. Services are composed in terms of already available atomic services, which are orchestrated in the service composition.

Service requests are used for service discovery, matching and composition. Service requests allow end-users or service developers to specify what they want the service to do for them, abstracting from the way this service is implemented, possibly in terms of a composition of atomic services. In SPICE we are developing the Service Creation Environment, which should create service compositions that support the service requested by an end-user. In order to obtain these compositions automatically, the service request and service descriptions of atomic and composite services need to be annotated with semantics, by using ontologies. Web services [2] are basic building blocks for the realization of services, but they lack semantics. Semantic Web [5] is an effort that provides service descriptions with semantics, which enables automatic reasoning on these descriptions. OWL-S [6] and SPATEL [4] allow the definition and creation of semantic annotated (web) services, using ontologies. These technologies are expected to enable automatic service composition.

A scenario to illustrate automatic service composition is the following: Bob wants to send a happy birthday message in Italian to Monica by SMS. He does not speak Italian so he has to use a dictionary in order to be able to write the message. Imagining that Bob has access to the SPICE platform or another platform that supports automatic service composition, he may issue to the platform the command *send "Happy Birthday my dear!" translated in Italian to +393123456789*. In this case it is highly probable that there is no single service to accomplish this task, so the platform attempts to find an appropriate service composition for that. Two services may have to be used, namely a translator and an SMS messaging service. This process relieves Bob from the hassle of manually discovering each required service and invoking these services.

3 SPICE Automatic Service Composition Engine

The SPICE Automatic Service Composition Engine (ACE) contains four basic components: *Semantic Analyzer*, *Composition Factory*, *Property Aggregator* and *Matcher*. Fig. 1 depicts the ACE architecture.

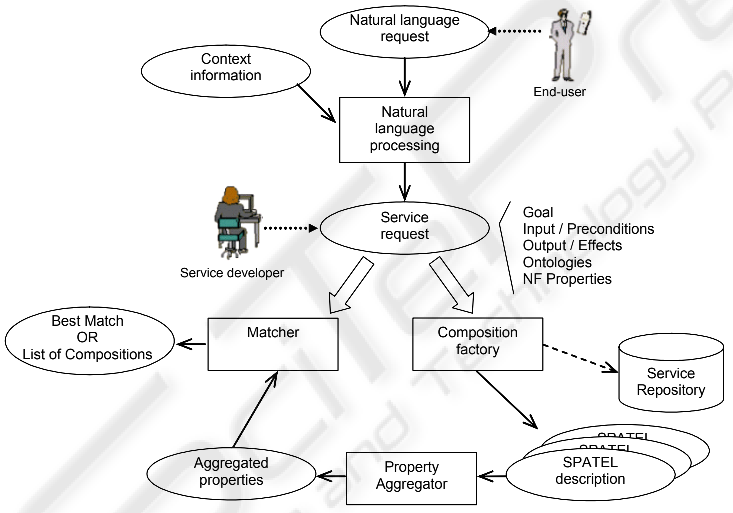

Fig. 1. SPICE ACE architecture.

Fig. 1 represents the two basic ACE usage scenarios: an end-user issues a service request in natural language, and a service developer issues a service request in some well defined formalism. The end-user is shielded from the system's complexity by requesting services in natural language. These requests are processed by the Semantic Analyzer, which constructs a formal service request according to the ACE's service request formalism, which is the same formalism used by the service developer.

When a service request is defined, the Composition Factory queries the service repository for a service that matches this service request. If such a match exists, the matching service is returned. In case no match is found, the Composition Factory creates a composite service that resolves the request. In general the Composition

Factory may generate multiple alternative compositions that match the service request.

Factory may generate multiple alternative compositions that match the service request.

Services and service requests are characterized by their functional and non-functional properties. Functional properties are the services' goals, inputs, outputs, preconditions and effects. These properties are used to perform the service discovery, matching and composition. Examples of non-functional properties are cost, security, performance, reliability, etc. Non-functional properties are used to limit the space of compositions that fulfil the service request, and to rank the generated set of compositions. Service and service request descriptions contain the functional and non-functional properties and the ontologies used to define these properties.

The compositions produced by the Composition Factory are passed to the Property Aggregator component, which computes the non-functional properties of the resulting compositions, by aggregating the non-functional properties of the atomic component services.

The set of generated composite services is then passed to the Matcher component. This component performs a matching between the composed and requested services, using their non-functional properties. In the end-user's use case, the best matching is returned to the end-user. This matching is obtained by taking the user profile and context information into consideration, which are managed by the SPICE platform. For a developer's request, several compositions may be returned. The developer can select the one that best fits his needs, possibly adapting it to fit more specific needs.

4 Semantic Graph-Based Composition of Web Services

The Composition Factory takes a formal request from an end-user or a service developer, and tries to find a service composition that matches the service request. In case a single service that matches the service request already exists, this service is returned as result.

4.1 Algorithm

In ACE, a formal service request contains the following elements: inputs, outputs, preconditions, effects, goals, non-functional properties and a list of domain ontologies. These elements are defined in OWL [15], and are used to discover, match and compose services. Available services are specified in SPATEL, and provided with semantic annotations similar to the elements above. This allows these services to be discovered through the goals of a service request, and then composed with other services by matching their interfaces in terms of inputs, outputs, preconditions and effects. In this paper we omit preconditions and effects in order to simplify the presentation of the algorithm. We expect that the addition of these elements to the algorithm is straightforward, since they can be seen as special cases of inputs and outputs, respectively.

The Composition Factory obtains service compositions by composing services according to a graph-based algorithm. The composition is created using an approach that starts with the outputs and possibly effects, and works backwards in the direction of the inputs and possibly preconditions. This implies that semantic descriptions of goals, inputs and outputs are compulsory for the ACE otherwise service discovery, matching and composition is not possible.

The algorithm holds a set of nodes *N* to be processed. Each element of *N* represents a set outputs *o* and goals *g* that do not match the inputs of the service request. The algorithm starts with a node n_0 representing the outputs and goals of the original service request o_0 and g_0, and issues a query for all the services that provide outputs o_0, support goals g_0 and requires the same inputs as the service request (i_0). In case a service that matches this query is found, the set of nodes *N* becomes empty and the algorithm can stop. In case the query returns a service s_1 that supports part of the goals g_0, delivers outputs o_0 but does not match the inputs i_0, the remaining goals g_1 are stored in the set of nodes *N* with the remaining inputs i_1 of the found service with respect to i_0 indicated as outputs. The algorithm then processes each node n_i by querying the service repository for services that match the goal g_i and the outputs o_i, and either decides to stop on this branch, or add more nodes to *N*, depending on the result of the query.

Fig. 2 shows the steps of our composition algorithm.

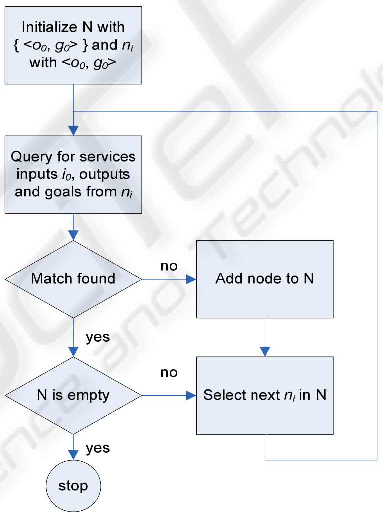
Initialize N with { <o_0, g_0> } and n_i with <o_0, g_0>
Query for services inputs i_0, outputs and goals from n_i
Match found
no
Add node to N
yes
N is empty
no
Select next n_i in N
yes
stop

Fig. 2. Service composition algorithm.

The algorithm delivers a composition graph as result, with possibly several branches representing alternative service compositions going from the requested inputs i_0 to the requested outputs o_0 and covering the requested goals g_0. The algorithm can execute indefinitely if matches are not found, or may result in compositions of too many component services. This implies that some heuristics must be defined to limit the algorithm execution and yet deliver useful results. Some possibilities are:

– stop at some graph depth, i.e., when the branches reach a certain number of edges. Since branches represent compositions, these heuristics limit the number of services in a composition;
– use non-functional properties of the composed service to select compositions that comply to the service request. This requires the calculation of the non-functional properties of the composed services, which can be done by aggregating the non-functional properties of the component services. By selecting only compositions that comply with non-functional properties, again the number of services in a composition is limited.

A measure of *semantic similarity* [7, 8] can also be used to determine the semantic distance between the query and the resulting services. This allows the algorithm to generate composition graphs with alternative semantically close branches, which can be useful in case no perfect match is found.

4.2 Example

We present an example to illustrate our automatic service composition algorithm. Considering the example in which Bob wants to send an SMS message in Italian to Monica (see section 2). Bob specifies its request in natural language as: *send "Happy Birthday my dear!" translated in Italian to +393123456789*. Since Bob is using the SPICE platform, a concrete service composition is going to be constructed at runtime, in order to cope with Bob's natural language service request. The ACE's Semantic Analyser extracts the desired service properties, creates a formal service request and passes it to the Composition Factory component. The formal service request can be represented as:

```
<Input>
        <"Translation:Language" name="sourceLanguage">
        <"Translation:Language" name="targetLanguage">
        <"Translation:Text" name="textToTranslate">
        <"Mobile:Telephone" name="destNumber">
</Input>
<Output/>
<Goal>
        <"Goal:translate">
        <"Goal:sendSMS">
</Goal>
<Non-functional>
        <"Latency:Response" value=5>
</Non-functional>
<Ontologies>
        <Goal Mobile Latency Translation IOTypes>
</Ontologies>
```

Fig.3 shows part of the Translation ontology that we assume in this example.

The Composition Factory takes this request and queries for a service that matches the inputs, outputs and goals. Suppose no service matches these elements, but the query returns the service with a sendSMS goal, matching `Goal:sendSMS` and no outputs, matching the outputs of the original service request. The sendSMS service is added to the composition graph, and a node representing the inputs of this service and the remaining (unsolved) goal is added to the set of nodes *N*. The inputs of the sendSMS service are destNumber and the text message to be sent. destNumber is of type Mobile:Telephone and corresponds to the telephone number given in the natural language request. The text message does not match the original input text of the service request since it is not in Italian. This means that a service with goal `Goal:translate` is necessary to provide an output of the type Translation:Text that is in Italian. Fig. 3, shows an excerpt of a Translation ontology that can be defined for this example. This ontology relates `Text` to the Language in which it is written. The translation service (not explicitly specified here) can translate text in one (source) language to text written in another (target) language. In this concrete example the translation is from English to Italian, which corresponds to the sourceLanguage and targetLanguage, respectively.

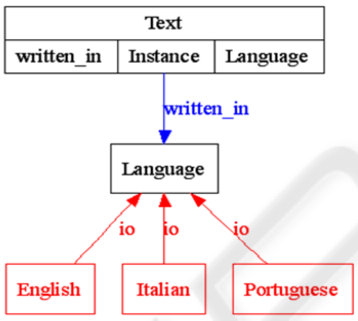

Fig. 3. Excerpt of the Translation ontology for our example.

We assume that the Translation service is found as a result of querying for a service that supports the `Goal:translate` goal and supports English and Italian as inputs for sourceLanguage and targetLanguage (the current node n_i). In this way the inputs of the original service request are completely resolved, and the composition process can stop, resulting on a composition of the services Translation and sendSMS.

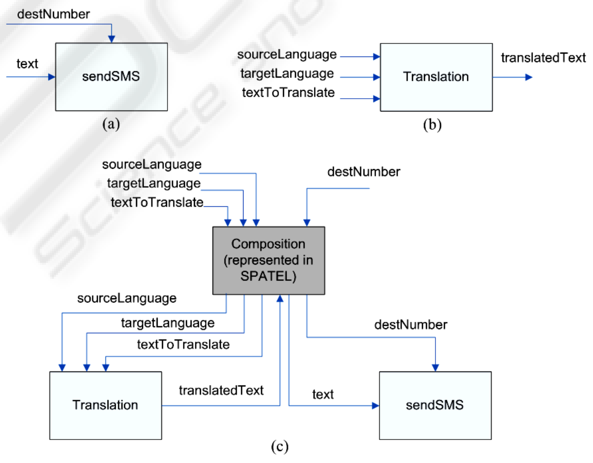

Fig. 4. Service composition example.

Fig. 4 depicts the composition process in terms of the services obtained throughout the process. Fig. 4 (c) shows the resulting composition. In SPICE, compositions are represented as SPATEL specifications, which could be translated to BPEL or executed directly by a SPATEL engine (currently being developed).

5 Related Work

In this section we present a brief overview of some techniques that cope with automatic service composition. For a detailed survey we refer to [9, 10]. We consider two techniques, namely graph-based and interface-matching automatic composition.

In [11], the authors propose a graph-based approach that constructs a composite service out of atomic services in case no single atomic service can satisfy a request. OWL-S [6] is used to describe the web services, in terms of inputs, outputs and execution workflow. A formalism and modelling tool called "interface automata" [12] is used to represent web services' information and perform compositions. Atomic services are stored in the repository as a graph, where nodes represent input and output parameters and edges represent web services. Each web service contains a description of its inputs, outputs and a dependency set of other web services. Given the graph, the previous information is used to discover the compositions that satisfy the request. The compositions can be constructed using four basic operations, namely concatenation, conditional structure, parallel structure and loop structure. When the compositions that match the request are discovered on the graph, they are passed to the interface automata tool, which performs the composition of the service based on the defined operation structures. If several alternative compositions are found, no mechanism for optimal selection is provided. There are no stop conditions either, which may complicate the search when several compositions do match the request.

The Interface-Matching Automatic (IMA) composition [13] aims at the generation of composite services by capturing expected service outcomes when a set of inputs is provided by the user. The result is a sequence of atomic services, whose combined execution achieves the user goals. Semantic web techniques are used to specify service semantics. Terms and concepts such as inputs, outputs and goals are described using the DAML-S service ontology [14]. Having this representation it is possible to proceed to the construction of composite services. The IMA service composition technique extracts inputs, outputs and constraints from the user request and navigates through the ontology to find the service sequences that match the user's input. After that, it chains services until they deliver the expected output. The goal is to find a composition that produces the best match within the shortest path in the graph, by using the notion of semantic similarity as matching metrics. Our proposed algorithm can be considered as an extension of this algorithm for what concerns the use of the non-functional constraints to select compositions.

6 Conclusions and Future Work

In this paper we present our initial proposal for the automatic service composition algorithm of the SPICE project. Fully automated composition is still an open research issue, and not many concrete results have been achieved in this area yet. By assigning semantic annotations to services, reasoners can be applied to automate the composition process to some extent. The application of these techniques to solve realistic service composition problems is yet to be assessed.

Our approach is based on semantic graph composition, and considers that service requests and service descriptions define functional and non-functional properties of required and offered services, respectively. We propose an algorithm for the Composition Factory that uses service goals and input/output matching to perform service composition. We assume the availability of a repository, organized according to domain ontologies. The Composition Factory is in principle capable of creating different alternative compositions that match the service request. The algorithm has been explained and a simple example has been used to illustrate its basic operation.

Our algorithm is still under development, and some improvements are expected to take place. Once we have a prototype, we expect to explore and improve several points of our algorithm, namely: the optimization of the composition process, the use of similarity measures in case no composition fulfils completely a service request; and the possible storage of service compositions created before for using in new compositions.

Acknowledgements

This work is supported by the European IST-SPICE project (IST-027617) and Dutch Freeband A-MUSE (Architectural Modelling for Service Enabling in Freeband) project (BSIK 03025).

References

1. Christophe Cordier et al.: Addressing the Challenges of Beyond 3G Service Delivery: the SPICE Platform. In: Workshop on Applications and Services in Wireless Networks (ASWN'2006). May 2006.
2. D. Booth et al.: Web Services Architecture. http://www.w3.org/TR/ws-arch, W3C Workinggroup Note. February 2004.
3. T. Erl: Service-Oriented Architecture (SOA): Concepts, Technology and Design. Prentice Hall, 2005.
4. J. P. Almeida, A. Baravaglio, M. Belaunde, P. Falcarin, E. Kovacs: Service Creation in the SPICE Service Platform. In: Wireless World Research Forum meeting on "Serving and Managing users in a heterogeneous environment", November 2006.
5. Semantic Web, http://w3.org/2001/sw.
6. David Martin et al.: OWL-S: Semantic Markup for Web Services. http://w3.org/Submission/OWL-S. November 2004.

7. K. Fujii and T. Suda: Dynamic service composition using semantic information. In: International Conference on Service Oriented Computing (ICSOC'04), November 2004, pp. 39-48.
8. F. Lécué, A. Léger: Semantic Web Service Composition Based on a Closed World Assumption. In: European Conference on Web Services (ECOWS'04), December 2006, pp. 233-242.
9. A. Alamri et al.: Classification of the state-of-the-art dynamic web services composition. In: International Journal of Web and Grid Services 2006 - Vol. 2, pp. 148-166.
10. J. Rao, X. Su: A Survey of Automated Web Service Composition Methods. In: Semantic Web Services and Web Process Composition (SWSWPC'04), July 2004, pp. 43-54.
11. S. V. Hashemian and F. Mavaddat: A Graph-Based Approach to Web Services Composition. In: Symposium on Applications and the Internet (SAINT'05), January 2005, pp. 183-189.
12. L. Alfaro and T. Henzinger: Interface automata. In: Symposium on Foundations of Software Engineering (FSE'2001), September 2001, pp. 109-120.
13. R. Zhang et al., Automatic Composition of Semantic Web Services. In: International Conference on Web Services (ICWS '03), June 2003, pp. 38-41.
14. A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, et. al.: DAML-S: WS Description for the Semantic Web. In: International Semantic Web Conference (ISWC '02), June 2002, pp. 348-363.
15. M. K. Smith, C. Welty, D. L. McGuinness: OWL Web Ontology Language Guide, http://www.w3.org/TR/owl-guide/.