A reuse-based approach to the correct and automatic composition of web-services

Paola Inverardi
University of L'Aquila
Dep. Computer Science
Via Vetoio, 67100, L'Aquila (AQ), ITALY
inverard@di.univaq.it

Massimo Tivoli
University of L'Aquila
Dep. Computer Science
Via Vetoio, 67100, L'Aquila (AQ), ITALY
tivoli@di.univaq.it

ABSTRACT

Service oriented technologies, such as web services, can be considered one of the latest trends in distributed computing. Nowadays, the Internet arena is populated by an ever more increasing number of web services. This has led to the need for reuse-based approaches to the *automatic* construction of new services as a *correct* composition of existing ones. A composition of services is correct when it respects the *Service Level Specification (SLS)* specified for the composite service to be built. To this end, we propose an extension of our previous work on the automatic component assembly. The aim of this extension is to propose an automatic and specification-based approach for constructing composite services from existing ones, which are discovered from a service repository. We instantiate the proposal in the context of web services. This work has to be considered as an ongoing work. In this paper, we also raise some research questions concerning the current gap between what we are able to do by means of the proposed extension and what still remains an open issue.

1. INTRODUCTION

Service oriented computing is considered as the paradigm that will support the new generation of global computers (i.e., the web, the grid), enabling the flexible interconnection of autonomously developed and operated applications [1]. This new paradigm leads to take into account a new approach to software development. That is, a service is not directly provisioned to its clients (as they refer it at design-time) but its provisioning is the result of a discovery and negotiation process that takes place at run-time. In this paradigm, services are usually understood as autonomous computational entities that can be described, published, requested, and dynamically composed over the Internet.

Nowadays, the Internet arena is populated by an ever more increasing number of web services. This has led to the need for reuse-based approaches to the *automatic* construction of new services as a *correct* composition of existing ones. A composition of services is correct when it respects the *Service Level Specification (SLS)* specified for the composite service to be built. Web service composition has attracted attention from both industry and the academic communities. BPEL4WS [7] (or simply BPEL) is an XML-based programming language that can compose web services manually.

In this paper, we propose an approach to automatically and correctly compose existing services in order to form a new (composite) service. The proposal is based on an extension of our previous work on the automatic component assembly [2, 8], implemented in the SYNTHESIS tool [3]. The aim of this extension is to propose an automatic and specification-based approach for constructing composite services from existing ones, which are discovered from a service repository. We instantiate the proposal in the context of web services.

Our approach assumes the existence of a centralized repository, such as the UDDI registry [13], for already implemented web-services published over the Internet. Each existing service, registered in the repository, publishes its SLS. The SLS defines what the service is and not how it is implemented. It is given in terms of the service signature, i.e., its WSDL [12], plus possible *invariants* defined over the signature's operations. An invariant specifies a precise business logic (i.e., a combination of operation invocations) that must be respected, by a partner, to correctly interact with the specified service. For example, it is not possible to perform any operation if, before, a *login* operation has not been performed. In practice, this business logic is usually given in form of a BPEL description of the service. As it has been done for specifying *component coordination policies* [3] in SYNTHESIS, within our method, a service invariant is modeled by means of a *Labelled Transition System (LTS)* with an expressive syntax for easily specifying quite complex invariants. Note that, such LTS-based behavioral description can be derived, e.g., from the BPEL description of the business logic of the service. Taking into account the previous assumption, our method takes as input the SLS of the composite service to be built. As it is done for the existing services in the repository, this SLS is a signature (i.e., the set of operation signatures of the composite service) plus invariants on the operations. Differently from an already implemented service (published in the repository), these invariants are partial (they do not model the whole BPEL process specifying the business logic of the composite service). That is, they partially specify the business logic of the composite service in terms of only a correspondence

ESSPE '07 September 4, 2007, Dubrovnik, Croatia

between an operation invocation on the composite service (e.g., a `<receive>` activity in BPEL) and a combination of operation invocations on the constituent services (e.g., a `<sequence>`, `<switch>`, `<pick>`, `<flow>`, or `<while>` activity composition of `<invoke>` activities in BPEL). What is missing is the interaction protocol that a customer of the composite service has to respect in order to correctly use it.

Note that, on one hand, simply giving a correspondence between a composite service operation and constituent services operations does not guarantee that when one uses the composite service, by performing invocations to its operations in any order, a business logic violation does not occur (e.g., deadlock, or more simply, the invocation of an operation in a state that is not consistent with respect to the service's business logic). On the other hand, it is realistic to assume that, in reusing existing services, the developer of the composite service has only the sufficient knowledge to make an operation of the composite service correspond to operations of the constituent services and he cannot guarantee the correctness of a more global business logic specification.

The aim of our method is to extend, modify, or partially reuse SYNTHESIS in order to automatically derive, from the SLS of the existing services and the (partial) SLS of the composite service, the correct (with respect to the deadlock-freeneess and the business logic of the constituent services) BPEL code of the composite service hence providing a concrete and correct specification of its business logic in terms of interaction with existing services.

This work has to be considered as an ongoing work hence addressing only a part of the underlining overall problem. For example, semantic or extra-functional properties of the composite service are not taken into account and their treatment is considered as future work. In this paper we will also raise unanswered questions concerning the overall actualization of our approach.

The remainder of the paper is organized as follows. Section 2 describes, as background, our SYNTHESIS tool for the automatic and correct component assembly whose adaptation to web services is proposed in this paper. Section 3 informally describes, by means of an explanatory example, our reuse-based approach to the correct and automatic composition of web services. Section 4 draws some conclusions, present few related works, and raises unanswered questions by discussing also possible future directions.

2. THE SYNTHESIS TOOL

This section introduces SYNTHESIS, a tool for automatically assembling correct and distributed component-based systems out of a set of already implemented black-box components [3]. SYNTHESIS implements our adaptor-based approaches to component assembly (see [2, 8]).

The implemented approach automatically generates a *distributed* adaptor for a set of black-box components. It is a specification-based and decentralized approach. Namely, given (a) the *interface* specification (i.e., the IDL) of each component, (b) the specification of the *interaction behaviour* of each component with its environment, and (c) a specification of the *desired behaviour* that the system to be composed must exhibit, it generates a set of *component wrappers* (one for each component). These wrappers suitably communicate in order to avoid possible deadlocks and enforce the specified desired interactions. They constitute the distributed adaptor for the given set of black-box components.

More precisely, starting from the components' IDL and from the specification of the components' interaction behaviour, SYNTHESIS automatically builds a behavioural model, i.e., an LTS, of a centralized *glue adaptor*. This is done by performing a part of the synthesis algorithm described in [8]. At this stage, the adaptor is built simply for modeling all the possible component interactions. It acts as a simple router and each request/notification it receives is strictly delegated to the right component. By taking into account the specification of the desired behaviour that the composed system must exhibit, SYNTHESIS explores the centralized adaptor LTS in order to find those states leading to deadlocks or violating the specified desired behaviour. This process is used to automatically derive the set of component wrappers that constitute the *correct*[1] and *distributed* version of the centralized adaptor.

SYNTHESIS supports two possible implementations of the generated distributed adaptor. One implements the adaptor's actual code as a set of COM/DCOM component wrappers (one for each component). These wrappers act on the component registries in order to interpose themselves among the components, intercept their messages, and coordinate them as it has been specified. The other implements the adaptor as a set of EJB component wrappers. Each wrapper is developed by using *AspectJ* to intercept the component messages and correctly coordinate them.

The method implemented by the current version of SYNTHESIS assumes the following data as inputs: (a) the interface specification of the components forming the system to be built. It is given as a set of IDL files, one for each component, implementing a *server* logic; (b) the specification of the desired behaviours that the system to be built must exhibit. It is given in terms of a set of LTSs with a specific syntax for the transition labels; (c) for each component (either client or server), the specification of its interaction protocol with the expected environment. It is an XML file that encodes an *high-level Message Sequence Chart* (hMSC).

These three inputs are then processed in two main steps: (1) by taking into account the inputs (a) and (c), SYNTHESIS automatically derives the LTSs that model the component interaction behaviour with the expected environment. From a component LTS, a partial model of the centralized glue adaptor is automatically built. It is partial since it reflects the expectation of a single component. By "unifying" all these partial adaptor models, SYNTHESIS automatically derives the LTS K of the centralized glue adaptor. (2) After K has been generated, SYNTHESIS explores it looking for those states representing the *last chance* before incurring in an execution path that leads to a deadlock. The enforcement of a specified desired behaviour is realized by visiting the LTS modeling it (input (b)). The aim is to split and distribute this LTS in such a way that each component wrapper knows which actions the wrapped component is allowed to execute. The sets of last chance states and *allowed actions* are stored and, subsequently, used by the component wrappers as basis for correctly exchanging *synchronizing information* when strictly needed. In other words, the generated component wrappers interact with each other to perform the correct behaviour of K with respect to deadlock-freeness and the specified desired behaviours.

[1]With respect to deadlock-freeness and the specified desired behaviour.

3. METHOD DESCRIPTION

In this section, by means of an explanatory example, we informally describe our method to the correct and automatic composition of web services. In Figure 1 we show our method by pointing out the input and output data processed, and the intermediary models that are produced and processed for synthesis purposes.

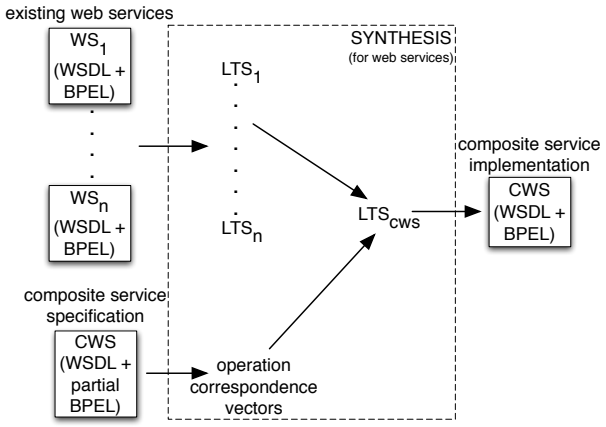

Figure 1: SYNTHESIS for Web Services

As already said in Section 1, the SYNTHESIS version for web services proposed in this paper, takes as input the SLS (i.e., WSDL + BPEL) of the existing services registered in the centralized service repository. Furthermore, SYNTHESIS for web services, takes as input also the partial SLS of the service to be built from the existing ones (i.e., WSDL + partial BPEL). From these two inputs, our method automatically derives the LTSs modelling the specified business logic for the existing services. For example, let us consider that we want to compose two existing web services, `LIB` and `PAY`, in order to build a composite web service `CWS` that implements an electronic library. Through `CWS`, an authorized customer can search for a book, order it, and pay for its order. `LIB` provides the capabilities of customer authentication and electronic library. `PAY` provides the capabilities of customer authentication and on-line payment. The following code is a fragment of the SLS of `PAY`. The first fragment has been taken from the WSDL specification of `PAY`, the second one from its BPEL process specification.

```
<definitions ...
  <portType name="PAY_PT">
    <operation name="login"> ...  </operation>
    <operation name="logout"> ...  </operation>
    <operation name="pay"> ...  </operation>
  </portType> ...
  <role name="PAY">
    <portType name="PAY_PT"/>
  </role>
  <service name="PAY_BP"/>
</definitions>

<process name="PAY_PROCESS" ...
  <partners>
    <partner name="customer" ...  />
    <partner name="book_vendor" ...  />
  </partners> ...
  <sequence>
    <receive name="authentication" partner="customer"
      portType="PAY_PT" operation="login" .../>
    <while ...> ...
       <receive name="payment" partner="customer"
        portType="PAY_PT" operation="pay" .../>
    </while>
    <receive name="exit" partner="customer"
      portType="PAY_PT" operation="logout" .../>
  </sequence>
</process>
```

SYNTHESIS for web services parses this SLS and automatically builds the LTS of `PAY` as it is shown in the right-hand side of Figure 2. The LTS of `LIB` (shown in the left-hand side of Figure 2) is built analogously to what has been done for `PAY`.

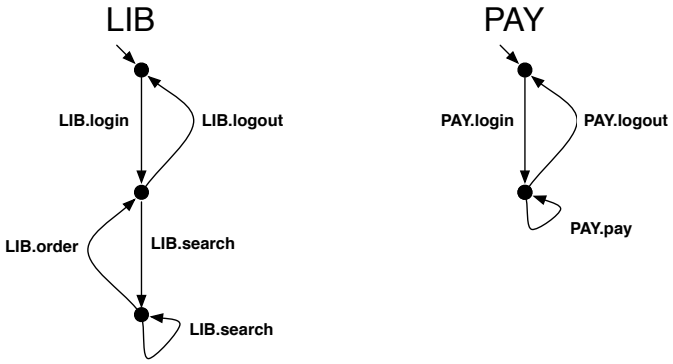

Figure 2: LTSs for `LIB` and `PAY`

The SLS of `CWS` defines a WSDL declaring three operations: `CWS.login` (i.e., customer login), `CWS.logout` (i.e., customer logout), and `CWS.getBook` (i.e., searching, ordering, and payment for a book). Moreover, its partial BPEL specification allows SYNTHESIS for web services to derive the following operation correspondence vectors:

```
CWS.login ::= LIB.login | PAY.login
CWS.logout ::= LIB.logout | PAY.logout
CWS.getBook ::= LIB.search -> LIB.order -> PAY.pay
```

The set of operation correspondence vectors (i.e., the set of "`::=`" equations) partially defines the business logic of the specified composite service.

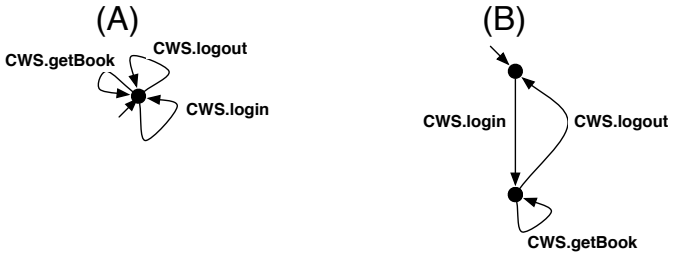

Figure 3: (A) most permissive LTS for `CWS` (B) correct LTS for `CWS`

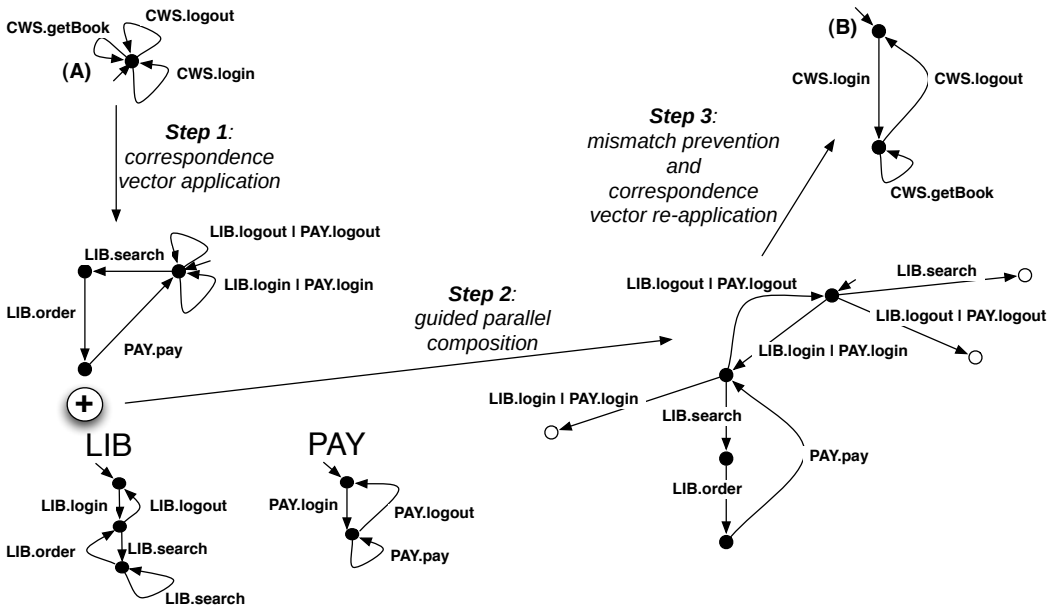

Figure 4: Method steps

In fact, it says nothing about the interaction protocol that must be respected by a customer when he invokes the composite service operations. It just specifies what execution flow (in terms of operations of the constituent services) is triggered once an operation of the composite service is invoked. By continuing our explanatory example, when `CWS.login` is invoked on the composite service, `LIB.login` and `PAY.login` are performed independently as two parallel activities (i.e., the "|" operator). It is done analogously for `CWS.logout`. When `CWS.getBook` is invoked, `LIB.search`, `LIB.order`, and `PAY.pay` are sequentially performed (i.e., the "`->`" operator).

From the WSDL specification of `CWS`, SYNTHESIS for web services trivially derives an LTS modelling the most permissive business logic for `CWS`, i.e., the one that allows a customer to invoke the `CWS`'s operations in any order and in any execution state (see Figure 3.(A)). It can be seen as the model of an *uncontrolled orchestrator* for `LIB` and `PAY` with respect to the operation "entry-points" of `CWS`. The uncontrolled orchestrator is not the correct business logic for `CWS`, although, e.g., no deadlock occurs. In fact, one can, e.g., requires to pay for an empty order that, although non-critical, represents a violation of the indeed business logic of `CWS`. To derive the correct business logic, and hence automatically complete the partially specified SLS of `CWS`, our method takes into account the LTSs of `LIB` and `PAY`, and combines them with the three operation correspondence vectors above specified. This is done in order to refine the LTS shown in Figure 3.(A) and automatically synthesize a *controlled* orchestrator for `LIB` and `PAY` with respect to `CWS`, whose LTS is shown in Figure 3.(B).

The LTS shown in Figure 3.(B) models the correct and complete business logic for `CWS`. From it, SYNTHESIS for web services, automatically derives the actual BPEL code for `CWS` hence automatically completing its partially specified SLS.

In Figure 4, we show the three steps automatically performed by SYNTHESIS to derive the LTS of `CWS`, which models its whole correct business logic.

In the first step, from the most permissive LTS of `CWS`, a new LTS is automatically built by rewriting each transition according to the specified correspondence vectors. Let us denote this new LTS as `CWS`$'$. Then, in the second step, a kind of LTS parallel composition operator is performed. This parallel composition takes into account the LTSs of `LIB`, `PAY`, and `CWS`$'$. Differently from a classical synchronous parallel composition operator between LTSs, our parallel composition deals with "concurrent" actions. For instance, the action `LIB.login | PAY.login` of `CWS`$'$ must synchronize with both action `LIB.login` of `LIB` and action `PAY.login` of `PAY` hence letting concurrently evolve the LTS of `CWS`$'$, `LIB`, and `PAY`. Furthermore, our parallel composition is guided by `CWS`$'$ in the sense that it does not look at all possible non-deterministic choices. In other words, through this parallel composition, our method searches for all those traces of `CWS`$'$ that can synchronize with traces of `LIB` and `PAY`. When a synchronization does not occur, a transition leading to a sink state (i.e., a state without outgoing transitions) is produced (see the white-filled states shown in Figure 4). Each sink state corresponds to a safety violation of the business logic of `CWS`$'$ with respect to its constituent services (i.e., `LIB`, and `PAY`). The third step performs *backwards error propagation* in order to prevent the detected business logic mismatches (i.e., all the finite traces in the parallel composition are removed). Moreover, during this last step, the correspondence vectors are re-applied in order to suitably rewrite the actions in the parallel composition hence obtaining the correct LTS for the business logic of `CWS`.

4. CONCLUSIONS

In this paper we proposed an automatic approach to the correct composition of already implemented web services. The aim of this composition is to automatically and correctly build a new web service. The proposed approach can

be seen as an extension and/or adaptation of our existing tool SYNTHESIS. Currently, SYNTHESIS is not for automatic web service composition but for automatic component assembly that is a similar problem.

To perform the method described in this paper, we need to modify our SYNTHESIS tool as follows: (i) replace the IDL parser with a WSDL parser; (ii) replace the hMSC-to-LTS translator with a BPEL-to-LTS translator; (iii) modify the LTS unification algorithm in order to take into account the correspondence vectors and the new parallel composition operator (the backwards error propagation step remains the same); and (iv) replace the COM/DCOM and EJB model-to-code transformer with a BPEL model-to-code transformer.

As SYNTHESIS currently is, it might be easily extended to deal with automatic web services *choreography*. This is especially true for the version of SYNTHESIS that, given a set of EJB components (that can be considered as simple web service implementations), a behavioral specification for them (that is similar to consider WSDL + BPEL), and a specification of the coordination policy (that is similar to consider BPEL), automatically synthesizes a set of component filters, which cooperate in order to prevent deadlocks and to enforce the specified policy (each filter might be coded as a choreography layer by using BPEL or WSCI). Although this apparently easy extension of SYNTHESIS to the automatic web services *choreography*, in this paper we proposed something harder to achieve and hence more interesting for our purposes. It concerns automatic web services *orchestration*. Orchestration, differently from choreography, does not concern coordinating (in a distribute way) a set of web services to reach a global goal, but it concerns composing a set of web services to form a new (composite) service.

Many works have been proposed in the literature for the automatic service coordination/composition, see [6, 4, 5, 11] just to mention few ones. Most of them are more focused on automatic *choreography* (i.e., coordination). Furthermore, as far as we know, very few tools have been developed for automatic *orchestration* (i.e., composition) in practical industrial contexts (i.e., web services embedding W3C standards technologies such as WSDL, BPEL, SOAP).

As already said in Section 1, this paper must be considered as an ongoing work. We are still a little bit far from really having available SYNTHESIS for automatic web services composition. Open issues that have not been taken into account in this paper are related to discovery issues. What kind of semantic information do we have to specify to support the discovery process of the "most adequate" existing services? And how do we have to specify that? Furthermore, open issues also concern the possibility to deal with a more complete, and hence, more realistic SLS, which should take into account also QoS characteristics of the service to be built, semantic information for supporting the discovery process, resource-awareness or, in general, context-awareness.

Referring to the relevance of the proposed method with respect to pervasive service-oriented environments, the discovery process should be automatic. This can be achieved by suitably integrating, e.g., our approach with *automatic schema matching* approaches [9, 10]. Moreover, the approach should be performed at discovery-time, i.e., the composite service equipped with its partial specification wants to join the Internet arena and hence be published in the service registry. The discovery of its constituent services is performed. Then, the automatic orchestration method proposed in this paper is carried on and the complete and correct WSDL+BPEL description of the composite service is derived. Finally, it is published in the registry and the composite service is ready to be invoked by its customers.

5. ACKNOWLEDGMENTS

This work has been partially supported by the IST EU project PLASTIC (www.ist-plastic.org).

6. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, berlin Heidelberg New York, 2004.

[2] M. Autili, M. Flammini, P. Inverardi, A. Navarra, and M. Tivoli. Synthesis of concurrent and distributed adaptors for component-based systems. In *LNCS 4344*. 2006.

[3] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In *In proceedings of the International Conference on Software Engineering (ICSE 2007) - Research Tool Demos Track.*, 2007.

[4] A. Brogi, C. Canal, and E. Pimentel. Behavioural types for service integration: achievements and challenges. In *Electronic Notes in Theoretical Computer Science, Elsevier, ISSN 1571-0661*, 2004.

[5] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. In *Electronic Notes in Theoretical Computer Science, 105:73-94, Elsevier. ISSN 1571-0661*, 2004.

[6] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *In A. Dan and W. Lamersdorf, editors, Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC 06), Springer-Verlag LNCS vol. 4294, pages 27-39*, 2006.

[7] IBM. BPEL4WS, Business Process Execution Language for Web Services, version 1.1, http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf. 2003.

[8] P. Inverardi and M. Tivoli. Software Architecture for Correct Components Assembly. volume LNCS 2804. 2003.

[9] H. R. M. Nezhad, B. Benatallha, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW 2007 Web Services Track*, 2007.

[10] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB journal*, 10:334–350, 2001.

[11] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *ICWS'04, IEEE Computer Society Press, pages 43–51, San Diego, USA.*, 2004.

[12] W3C. Web Service Definition Language, http://www.w3.org/tr/wsdl.

[13] W3C. UDDI technical white paper, http://www.uddi.org/pubs/lru uddi technical paper.pdf. 2001.