A Dynamic Composition and Stubless Invocation Approach for Information-Providing Services

Federica Paganelli, *Member, IEEE*, and David Parlanti

***Abstract*— The automated specification and execution of composite services are important capabilities of service-oriented systems. In practice, service invocation is performed by client components (stubs) that are generated from service descriptions at design time. Several researchers have proposed mechanisms for late binding. They all require an object representation (e.g., Java classes) of the XML data types specified in service descriptions to be generated and meaningfully integrated in the client code at design time. However, the potential of dynamic composition can only be fully exploited if supported in the invocation phase by the capability of dynamically binding to services with previously unknown interfaces. In this work, we address this limitation by proposing a way of specifying and executing composite services, without resorting to previously compiled classes that represent XML data types. Semantic and structural properties encoded in service descriptions are exploited to implement a mechanism, based on the Graphplan algorithm, for the run-time specification of composite service plans. Composite services are then executed through the stubless invocation of constituent services. Stubless invocation is achieved by exploiting structural properties of service descriptions for the run-time generation of messages.**

***Index Terms*— Web Services, Semantic Web, Service Composition, Service Invocation, XML, Planning, Service Brokering, Service-Oriented Architecture.**

I. Introduction

THE Service-Oriented Architecture (SOA) is a widely adopted paradigm that eases the design and development of distributed applications across heterogeneous organizational domains. Loose coupling among service providers and consumers is a reference principle for SOA implementations. This principle emphasizes the need for reducing dependencies among the service implementation, its published contract, and service consumers [1]. A way of achieving loose coupling is to rely on the intermediation capabilities provided by a third-party component, usually called a Service Broker, especially for service discovery and composition. Given a service consumer request, the Service Broker specifies the workflow of service invocations that have to be executed to address the request. Composite services can be specified in business process languages, such as the Web Services Business Process Execution Language (WS-BPEL) [2] and the Business Process Model and Notation (BPMN) [3], and executed directly by the requester or a third-party component through existing workflow engines or ad-hoc applications.

Many researchers have proposed models and mechanisms for automating service discovery and composition, and enabling composite services specification at run-time (dynamic service composition) [4]. But fewer efforts have focused on the related topic of dynamic service invocation. In practice, service invocation is typically handled by client components (stubs) that are tightly bounded to service implementations, and are generated from the description of the service to be invoked [5]. The invocation of services with a similar interface requires a new stub component to be generated. The Web Service Description Language (WSDL) [6] is a standard specification for describing service interfaces, based on the eXtensible Markup Language (XML) [7].

Semantic Web technologies can help to identify similarities across service signatures, and build translation layers among similar service signatures. However, although reducing the need for client interfaces reprogramming, most existing approaches rely on a design-time step of programmatic binding to a reference service interface.

We argue that the full potential of dynamic composition and related tasks (e.g., dynamic service selection and substitution) can only be fully exploited if it is complemented by the capability of dynamically binding to services with previously unknown interfaces.

Several existing Web Services invocation frameworks (e.g., Apache WSIF [8], Apache Axis 2 [9], Codehaus XFire [10], Apache CXF [11], and Java API for XML-based Web Services [12]) provide some capabilities for late binding to Web Services (e.g., via dynamic proxying). A weakness of these solutions is that the client code is highly dependent upon specific toolkit APIs [13].

Buhler et al. [13] solved that problem by minimizing the dependency of the client code upon the adopted invocation

Manuscript received February 6, 2012, revised May 28, 2012 and October 31, 2012, accepted November 12, 2012. The Associate Editor coordinating the review of this paper and approving it for publication was J.P. Martin-Flatin.

F. Paganelli is with the National Interuniversity Consortium for Telecommunications, Italy (phone: +39 055 4796382; fax: +39 055 4796427; e-mail: federica.paganelli@unifi.it).

D. Parlanti is with Negentis, Firenze, Italy (e-mail: david.parlanti@gmail.com).

framework. Leitner et al. [5] allow clients to invoke a service by sending a request message to an intermediate component. The main weakness of these two approaches is the need for a preprocessing step to compile the target service description into an internal object representation of XML data types. This limitation is due to the fact that the client must marshal input data into a request message, and then unmarshal the returned data from the response message. This can easily be done by libraries that automate the generation of compatible classes that represent XML data types, but these classes have to be known at design time to be meaningfully exploited and integrated in the client code.

Our service composition and invocation approach addresses this issue by removing the need for generating an internal representation of XML data types at design time. Semantic and structural message properties encoded in service description files are exploited to implement a mechanism that allows the run-time specification of composite service plans, and their execution through the stubless invocation of constituent services. Semantic annotations are used to define relationships between elements of different service interfaces. Structural properties are exploited to define simple transformation rules (aggregation/disaggregation of XML elements). The joint use of these mechanisms allows the dynamic specification and execution of a sequence of service invocations, without the need for previously compiled classes that represent XML data types in service descriptions.

As a result, our approach for dynamic composition and stubless invocation of information-providing services truly enables loose coupling between service consumers and providers.

The remainder of this paper is organized as follows. Section II gives background information on SOA. Section III introduces the main principles of our approach. In Section IV, we describe our service profile model. Sections V and VI describe our approach for dynamic service composition and invocation. Section VII reports on the implementation of our prototype. In Section VIII, we show test results. In Section IX, we discuss related work. Finally, Section X concludes the paper with insight for future work.

II. Background

According to Erl [1], the main principles underlying the SOA architecture are fivefold: service contract, loose coupling, service abstraction, service reusability, and service composability.

The *Service Contract* expresses the purpose and capabilities offered by services. It is mandatory for a service contract to express the technical interface details (e.g., data types, message structure, network protocols, and endpoints in WSDL documents). The *Loose Coupling* principle emphasizes the need for reducing dependencies among the service implementation, its published contract, and service consumers. It enables the design and evolution of a service implementation while minimizing the impact on service consumers' interfaces. *Service Abstraction* is a cross-cutting aspect of service-orientation that consists in hiding as much as possible low-level details of a service interface and implementation. The level of abstraction typically affects service contract granularity and the achieved loose-coupling degree. *Service Reusability* advocates that services should be designed for serving more than one consumer. The *Service Composability* principle expresses the need for designing services as building blocks that can be effectively used in future service compositions.

The SOA model traditionally encompasses three main roles [1]: the *Service Provider*, which exposes a given service and publishes the service description in a registry; the *Service Registry*, which offers service description publishing and querying capabilities; and the *Service Consumer*, which queries the registry and binds to a service endpoint when an operation is invoked. In practice, SOA systems have introduced a fourth intermediating role: the *Service Broker*.

In this context, we model dynamic and loosely coupled interactions among clients and services as a three-step process.

In the first step, called *Abstract Process Definition*, the request message is analyzed by the Service Broker, which decides whether the request can be handled by accessing the Service Registry knowledge base. There are three possibilities at this point: i) the request can be handled by invoking one service; ii) the request can be handled by invoking a specific flow of operations (i.e., a composite service); or iii) the request has no known solutions in the domain of registered services. In the first two cases, the Service Broker specifies the service invocations to be executed to achieve the client goal.

The second step, called *Concrete Composition Process Mapping*, consists in mapping an abstract process specification onto a set of executable actions, and selecting the concrete endpoints to be invoked.

The third step, *Service Execution*, concerns the invocation of a set of service endpoints. Relying on stub-based client interfaces for service invocation inevitably limits the scope of dynamicity of service composition, because stub-based interfaces are programmatically bounded to service interfaces and in practice they are generated at design time. Whereas stubless invocation can help to achieve run-time binding to services whose interface was unknown at design time.

III. Main Principles of Our Approach

The objective of this work is to provide a lightweight semantic and data-centric approach that enables the dynamic composition of services and the automatic run-time binding to service endpoints. We do so by re-interpreting and fully exploiting service-oriented principles, in order to favor loosely coupled interactions among service providers and consumers.

Our approach is based on two principles. First, we model a service as a message processor: its interface is modeled in terms of the messages it can transmit and receive [14]. Second, we rely on syntactic, structural, and semantic information that can be encoded in service descriptions for realizing dynamic service discovery, composition, and invocation.

These principles have been put into practice in the design of a model for representing service functional profiles. This model aims at providing a layer of abstraction over existing service descriptions in order to accommodate the representation of different service specifications, and to ease the mediation of heterogeneities at the level of protocols, data formats, and semantics.

By revisiting the data modeling layers adopted in database design, and by taking inspiration from Wilde's analysis [15] on Representational State Transfer (REST) interactions, we model the message exchange at the conceptual, logical, and physical levels.

At the *physical level*, data have to be serialized in properly formatted messages to be sent on the wire. Well-known examples include XML, the Simple Object Access Protocol (SOAP) [16], and the JavaScript Object Notation (JSON) [17]. Invocation is thus achieved by sending the message to the service endpoint over the proper communication protocol.

At the *logical level*, syntactic and structural rules model the logical structure of the target message and specify how data have to be embedded in a valid document. In this work, we exploit the capabilities of XML Schema [18] in modeling tree-based document structures, as it is a reference standard in the WSDL specification [6].

At the *conceptual level*, we explicitly represent concepts and relations that are embedded in the exchanged messages. Ontologies provide a formal representation of concepts and relations in a given domain. Ontology-based models and technologies can thus help to mediate and reconcile differences among service interfaces. As we cannot assume that service engineers will always refer to the same ontology, this is the level where semantic heterogeneities should be mediated. We do not address here the topic of ontology mediation, for it is already widely covered in the literature [19][20].

As mentioned above, we deal with XML documents whose grammar can be expressed in XML Schema. In valid XML documents, instance data values are ultimately carried by leaf nodes (i.e., attributes and simple type elements). We intentionally ignore mixed content models (i.e., a content structure where text data and subelements can be mixed in an element), as mixed content is not a best practice for interoperability [21]. In our model, Atom entities are used to represent leaf nodes and their properties. While some existing works [22][23] represent leaf nodes with a minimal set of syntactic properties (typically the local or qualified name of the XML node), we characterize the atoms with three types of properties. *Syntactic Properties* rely on the basic rules of the chosen formatting language (XML), such as the local or qualified name. *Structural Properties* (i.e., the hierarchical structure of the message) represent a context for a leaf node. *Semantic Properties* refer to semantic concepts and relations that describe an XML node.

While structural and syntactic similarities could be exploited for improving XML document comparison techniques [24], we assume here that elements with the same qualified name, but embedded in different messages, are semantically different. The meaning implied by the document node could be explicitly conveyed by a concept in a shared domain representation (i.e., an ontology). Semantic matching for similarities and equivalences may be inferred by reasoning on such semantic-level information [20].

We call our proposed approach "data-centric" because properties of atom data ultimately carried by instance messages are considered key entities in the model. Our approach is also "lightweight" because we rely on lightweight semantic annotations to enhance service descriptions with semantic properties.

We achieve loose coupling by using brokering services that exploit the expressiveness of the model. First, we assume that the client can invoke a service by sending a request message that specifies input parameters and the target output. When the Service Broker receives the request message, it interprets it and decides whether the client goal data can be obtained by using the functional capabilities offered by registered services. By exploiting structural and semantic descriptions of registered service profiles, the system tries to reach distributed target data atoms through dynamically computed "routing tables", starting from the data provided in input. Such "routing tables" describe a service invocation sequence that workflow engines or clients may execute to obtain the requested data.

In other words, analogously to existing planning-based approaches to service composition [4], we interpret the problem of solution search as the problem of finding a graph of functional profiles whose input data are known parameters within the query, and whose output (message) data contain the expected data (goal). Matching input and output data across services makes it possible to build a graph of connected functional profiles (see Fig. 1). Semantic properties can be exploited to connect different atoms referring to the same concept or to concepts related by some semantic relations.

As far as service discovery is concerned, full-fledged profiles enable authorized clients to submit complex queries to a Service Registry (i.e., by target message, data types, non-functional properties, and/or semantic annotations). Once an

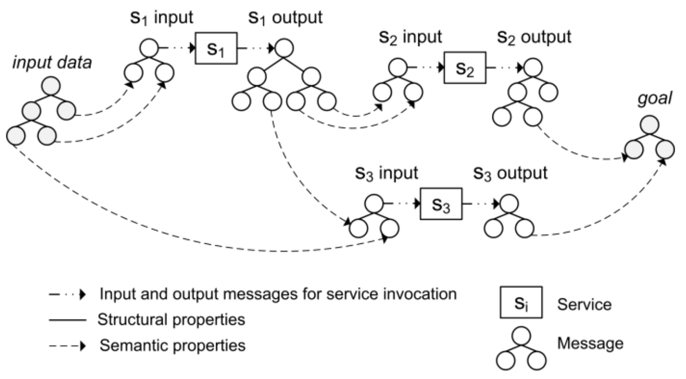


Fig. 1. Service composition through input/output matching

endpoint has been retrieved, a service invocation can be triggered by creating at run-time an XML message instance that is valid according to its structural rules, and then delivering it to the service endpoint over the protocol specified in the binding information.

Relying on atom structural and semantic properties enables the automation of the following actions for stubless service invocation: (i) matching instance data values (available in the client request or in response messages of previous invocations) with the atoms of the input message; (ii) creating an input message whose structure complies with the atoms' structural properties; and (iii) sending the input message over the proper communication channel to the target service endpoint. When the response message is received, the opposite operations are performed: (i) extracting the data values embedded in the message; and (ii) finding the proper associations with the atoms needed for the subsequent service invocations via semantic matchmaking.

Thanks to these mechanisms, clients can interact via message-based passing with a Service Broker and ask for some functional capabilities, independently of any interface details of the registered services. The Service Broker tries to find a solution, which is specified as a composition of service invocation and data manipulation operations. Such operations can be performed (by the Service Broker or directly by the client) via generic mechanisms (e.g., data aggregation and message delivery) that can be configured at run-time according to the information contained in the service profile.

Modeling a service as a message processor has the advantage of minimizing the requirements on service description complexity, as we rely on input/output stateless service signatures rather than on complex stateful signatures. This assumption limits the range of applicability of our planning-based composition approach to information-providing services that are exposed via stateless service signatures, as our message processor model cannot be applied to stateful services whose capabilities are described by specifying inputs, outputs, preconditions, and effects. This limitation does not apply to the dynamic invocation technique, which can be used as a utility for stubless service invocation by third-party software components.

IV. Service Profile Model

Our Service Profile Model defines modeling primitives for representing structural and semantic properties of service interfaces. The model is defined in terms of Service Entities, which represent functional and structural information of service interfaces, and Qualifying Attributes, which extend the model with additional properties such as non-functional or domain-specific attributes.

Fig. 2 depicts the main entities and properties of the model in the UML class diagram notation. Service Entities represent the basic constructs of a functional profile in terms of Operations, Messages, and Atoms. An Operation defines the functional capability of a service in terms of input/output message pairs. A Message is an XML document. An Atom represents an XML entity that carries data values in an input or output Message. In well-formed Messages, Atoms are XML attributes or simple type elements.

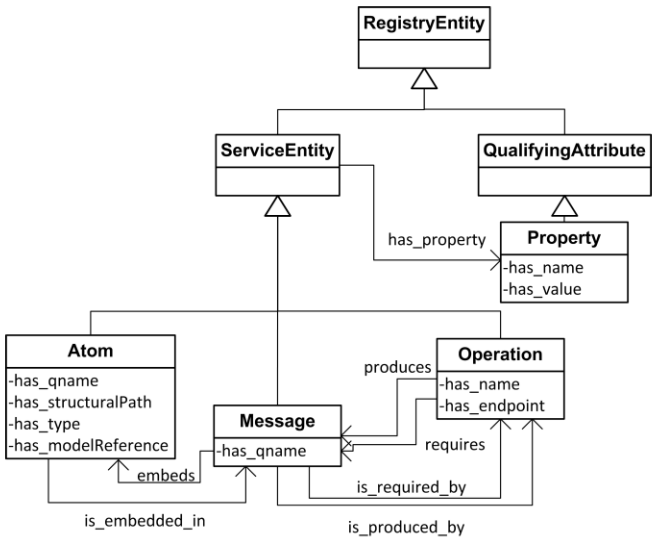


Fig. 2. Service Profile Model

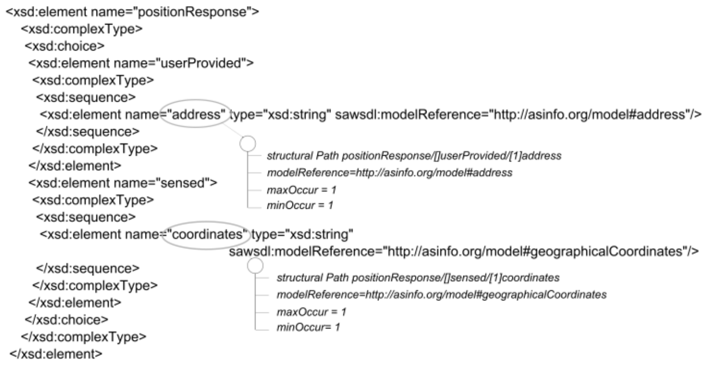
Atoms, Messages, and Operations are identified by a Qualified Name (*has_qname* attribute). An Operation may be provided by one or more endpoints (*has_endpoint* attribute).

Atoms are the key entities in our model. Each atom is characterized by a set of structural information, especially by a *structural path* (i.e., by its position inside the message structure). Due to the underlying XML information model, atom structural paths identify either element leaves or element attributes. The final association of atoms with optional *semantic properties* can be used to link ontological annotations to data embedded into each message.

Fig. 3 shows an example of properties that can be extracted from an XML Schema definition for a message document.

A. Structural Properties

An Atom is characterized by syntactic information (i.e., its qualified name) and two structural properties that specify how the atom is embedded in a message: (i) the *type* attribute, which specifies the atom data type (referring to the set of pre-defined types in XML Schema); and (ii) the



```
<xsd:element name="positionResponse">
  <xsd:complexType>
   <xsd:choice>
    <xsd:element name="userProvided">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="address" type="xsd:string" sawsdl:modelReference="http://asinfo.org/model#address"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
    <xsd:element name="sensed">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="coordinates" type="xsd:string"
                     sawsdl:modelReference="http://asinfo.org/model#geographicalCoordinates"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

structural Path positionResponse/[]userProvided/[1]address
modelReference=http://asinfo.org/model#address
maxOccur = 1
minOccur = 1

structural Path positionResponse/[]sensed/[1]coordinates
modelReference=http://asinfo.org/model#geographicalCoordinates
maxOccur = 1
minOccur= 1

Fig. 3. Example of atom properties extracted from an XML Schema document

has_structuralPath attribute, which specifies the position of the atom in the XML document tree by representing the chain of parent elements up to the document root and parents' content models. We defined a compact textual notation (called Structural Path Model) that encodes these structural constraints as a string.

Our Structural Path Model has some aspects in common with XPath [25], but different objectives. The primary purpose of XPath is to select nodes in an XML document. According to the XPath Data Model [26], XPath uses a path notation based on a compact non-XML syntax for navigating the hierarchical structure of an XML document. Conversely, our Structural Path Model embeds in a compact path notation the information needed for performing two tasks: (i) extracting a leaf node from an XML document; and (ii) assembling the available atoms in an XML document that has been validated against its XML Schema specification. The automated execution of these tasks is a necessary condition for enabling our mechanism for stubless invocation. To this end, a Structural Path Model expression specifies structural information that characterizes a branch of the XML document tree from the document root node to a target leaf node. The Structural Path Model can be viewed as a compact notation of the XML Schema specification for an XML Atom. The use of the Structural Path Model for service invocation is described in Section VI.

The grammar of a Structural Path Model expression is defined as follows:

```
<pathExpr> ::= <nodeName> / <order>
(<atomName> | <pathExpr>)
<order> ::= "[" [0-9]* "]"
<atomName> ::= <nodeName> | "@ " <nodeName>
```

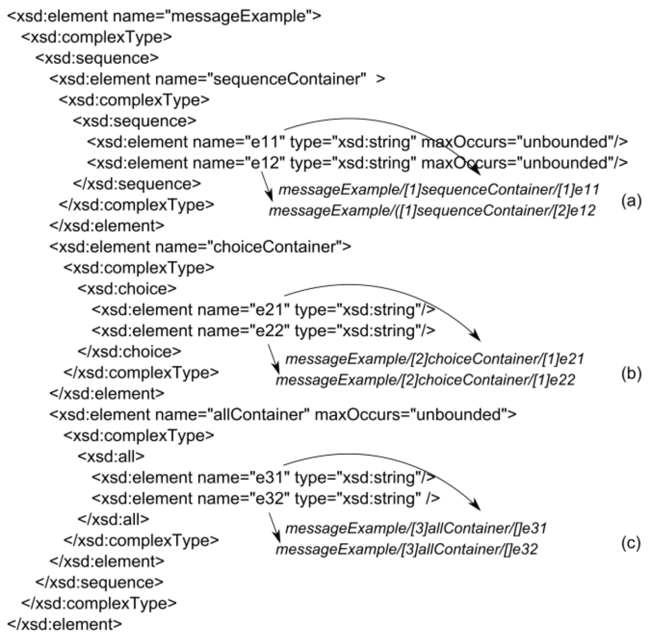
This grammar includes the following symbols: (i) the containment symbol "/" that represents a direct parent-child relationship between elements; (ii) the attribute reference symbol "@"; (iii) the <nodeName> symbol that represents the name of an XML node; and (iv) the ordering symbol "[…]" that is used to enforce a sequential order among siblings (the square bracket may be empty or contain an integer expressing the element ordering).

A Structural Path Model expression represents solely absolute paths. As a consequence, the first node in the path is always the root element of the message, and the path expression contains the whole hierarchical chain of parent-child nodes up to the target atom data. An atom structural path is thus a sequence of *p* node names separated by "/".

For instance, for $1 < i < p - 1$, if nodes n_i and n_{i+1} are separated by a "/", then n_{i+1} is a child element of n_i. The two expressions n_i / [1] n_{i+1} and n_i / [2] n_{i+2} mean that n_{i+1} and n_{i+2} are siblings and n_{i+1} must appear before n_{i+2} in instance documents.

The joint use of the container and ordering symbols enables us to express the most relevant structural constraints of XML Schema content models: sequence, all, and choice. In the *sequence* content model (i.e., when child elements must appear in the instance document in the same order as they are declared), the ordering symbol is not empty and is assigned with the proper sequential index (see expression (a) in Fig. 4). In the *choice* content model (i.e., when child elements exclude each other), the ordering symbols have the same value (see expression (b) in Fig. 4). In the *all* content model (i.e., when all elements can occur with zero or one multiplicity and can appear in any order), the ordering symbol is left empty (see expression (c) in Fig. 4).

Although it does not fully cover the whole XML Schema specification yet, our Structural Path Model does provide constructs for representing most of the structural properties that are typically encoded in service description files.



```
<xsd:element name="messageExample">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="sequenceContainer"  >
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="e11" type="xsd:string" maxOccurs="unbounded"/>
           <xsd:element name="e12" type="xsd:string" maxOccurs="unbounded"/>
         </xsd:sequence>
       </xsd:complexType>
      </xsd:element>
      <xsd:element name="choiceContainer">
        <xsd:complexType>
          <xsd:choice>
            <xsd:element name="e21" type="xsd:string"/>
            <xsd:element name="e22" type="xsd:string"/>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="allContainer" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="e31" type="xsd:string"/>
            <xsd:element name="e32" type="xsd:string" />
          </xsd:all>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

```
messageExample/[1]sequenceContainer/[1]e11
messageExample/([1]sequenceContainer/[2]e12      (a)

messageExample/[2]choiceContainer/[1]e21
messageExample/[2]choiceContainer/[1]e22         (b)

messageExample/[3]allContainer/[]e31
messageExample/[3]allContainer/[]e32             (c)
```

Fig. 4. Examples of Atom Structural Path expressions for XML Schema content models: sequence (a), choice (b) , all (c)

B. Semantic Properties

Semantic properties link service profile entities, especially atoms, with concepts defined in a given shared domain representation. Semantic properties allow the binding of structural atom properties to their intended meaning.

Relations among atom data can be inferred through semantic reasoning. This may help to decouple client and service provider interfaces, as differences in message structure and syntax may be reconciled at the level of semantics. This may also help to define composite services as sequences of service invocations, as it is possible to decide whether instance data values obtained in a response message can be used to build a valid request message for the next service invocation in the sequence.

In our Service Profile Model, the "*has_ modelReference*" property is used to represent references to semantic concepts. This type of information can be extracted from semantic service descriptions.

Although alternative models for semantic service descriptions exist, such as the Web Service Modeling Ontology (WSMO) [27]) and the Web Ontology Language for Services (OWL-S) [28], we rely on the Semantic Annotation for WSDL (SAWSDL) specification [29] because it is a W3C standard for lightweight annotations. SAWSDL extends WSDL with annotations pointing to semantics. Software systems can thus interpret and process these concept representations to automate tasks such as service discovery, composition, selection, negotiation, mediation, and invocation.

C. *Qualifying Properties*

Further optional properties (called Qualifying Properties) can be added to each input/output message pair to describe the non-functional aspects of service operations. For instance, Quality of Service (QoS) properties (e.g., availability, reliability, and reputation) and domain-specific attributes (e.g., geo-referenced information) can be represented as qualifying properties.

V. Dynamic Service Composition

Dynamic service composition deals with the problem of handling a client request by specifying a composite service that matches the request’s input and goal data and executing it at run-time.

Planning techniques from artificial intelligence can be exploited to find service composition solution plans at run-time [4]. A planning problem can be represented as a 5-tuple (S, S_0, G, A, Γ), where S is the set of all possible states of the world, $S_0 \subset S$ denotes the initial state of the world, $G \subset S$ denotes the goal state of the world, A is the set of actions that the planner can perform while attempting to change one state to another state of the world, and the translation relation $\Gamma \subseteq S \times A \times S$ defines the preconditions and effects for the execution of each action. When applying these concepts to the service composition problem, A represents the set of available services; S_0 and G are, respectively, the initial and goal states provided by the composite service requester; Γ denotes the state change function of each service.

To leverage the expressiveness of our Service Profile Model, we represent the service composition problem in a STRIPS model – a widely used model for representing planning problems. We use the Graphplan algorithm [30] to solve the planning problem (i.e., to verify if the given request has a feasible solution in the known domain, and to dynamically specify the service invocation flow). We chose the Graphplan algorithm for three reasons: it is a well-known planning algorithm for which several implementations exist; it is guaranteed to terminate when no valid plan exists; and it has a polynomial complexity (as opposed to the exponential complexity of exhaustive search [31]).

A. *Service Composition Problem as a STRIPS Model*

In this subsection, we describe the STRIPS model defined for handling the service composition problem.

STRIPS operators have preconditions, add-effects, and delete-effects that are represented as conjuncts of propositions, and have parameters that can be instantiated to objects in the world. Preconditions have to be valid immediately before the operator is applied. Add-effects and delete-effects are the sets of literals added to, or deleted from, the world state after the operator ends. An instantiated operator is called an action [32].

We use the STRIPS model to represent the profile of registered services. To this end, we define three types of operators: functional, structural, and semantic operators.

Functional operators represent service operations (i.e., functional capabilities). For each operation, input messages are modeled as preconditions, and output messages as add-effects. Messages are identified by their fully qualified name. An operation is univocally identified by the qualified names of input and output messages. The operator type is identified by the label *invoke_service*.

Structural operators represent structural containment relationships between messages and atom data. We distinguish two operator types: the *extract* operator that extracts atom data (add-effects) from the containing message (preconditions), and the *embed_into* operator that links atom data (preconditions) to a message (add-effects). For each atom to be embedded in or extracted from a message, a specific action of type *embed_into* or *extract_from* is instantiated.

Semantic operators encode the semantic properties that associate atoms with semantic concepts. These operators act as “semantic bridges” that connect different atoms via a common semantic concept. By applying semantic matchmaking techniques, it is possible to compute different degrees of matching between two concepts in an ontology, and consequently between output and input atom data in service messages. As discussed by Lécué et al. [20], there are five possible matching types between the output parameter o_i of a service s_i, the input parameter i_j of a service s_j, and the numerical values that express similarity:

- *Exact*: if o_i and i_j are equivalent (similarity value = 1).
- *Plugin*: if o_i is sub-concept of i_j (similarity value = 0.75).
- *Subsume*: if o_i is a super-concept of i_j (similarity value = 0.5).
- *Intersection*: if the intersection of o_i and i_j is satisfiable (similarity value = 0.25).
- *Disjoint*: o_i and i_j are incompatible (similarity value = 0).

Techniques that exploit all these semantic matchmaking degrees can dynamically infer meaningful service compositions that are rarely executable, due to unsolved structural and syntactic mismatches between XML messages [20]. Semantic relationships other than the equivalence (e.g., similarity or subsumption) may help to find abstract solution plans, but do not provide clear and safe hints for executing concrete solution plans. When several types of semantic relationships can be defined, we rely exclusively on the *Exact* matching degree. Thus, we use a single *is_equivalent_to* operator for binding atom data to their corresponding semantic concepts, as well as concepts to other equivalent concepts. In other words, we rely on an exact semantic equivalence in order to enable the transfer of a data value from one atom to another, as required when actually executing the service invocations flow.

B. Graphplan

Graphplan is a general-purpose planner that was proposed by Blum and Furst [30] to provide an effective way of building plans in STRIPS domains. The Graphplan algorithm compiles the problem into a structure called a *planning graph*. The planning graph is created in a forward direction from the initial conditions, and then expands itself one level at a time until a solution is found.

The first level contains the initial conditions. Each subsequent level has a node for each action that might possibly be performed (i.e., whose pre-conditions all exist in the previous level). At each level, the algorithm checks whether the propositions in the goal are all present at the current level. In that case, the algorithm searches for a valid plan in a backward-chaining manner.

A plan is valid if it satisfies the following conditions: the actions at the same level do not interfere (e.g., when an action deletes a precondition or an add-effect of another action); each action's preconditions are true at that point in the plan; and goals are satisfied at the end of the plan. If no valid plan exists at that level, the planning graph is expanded by adding another level.

When a solution is found, the Graphplan ends its search and returns the shortest feasible sequence of actions required to meet the goal. The Graphplan is guaranteed to terminate with a solution if a valid plan exists, or with no plan if the problem is unsolvable [30].

C. Dynamic Service Composition

The dynamic service composition process is threefold. First, the client request message is analyzed in order to extract input data and expected goals. In this step, semantic annotations embedded in the request message may be used to translate client request into concepts expressed in shared ontologies. Second, the problem specification is translated into a STRIPS model and forwarded to the planner. Third, if a feasible solution exists, the planner returns the specification of the set of invocations to be performed; otherwise it terminates after a finite number of steps, concluding that no solution exists.

Fig. 5 depicts a basic example in the application domain of maritime surveillance. The client wants to gather possible threats close to a given vessel (e.g., unidentified vessels nearby). The request message is thus made of two parts: the input parameters (the URLs for the ship identifier and the time interval concepts in a given ontology), and the output (the URL pointing to the unrecognized target concept). The solution plan depicted in Fig. 5 is a flow made of structural, semantic, and functional operators. The *vesselIdentifier* concept is mapped onto an atom (*vesselId*) via the *equivalent_to* operator. This atom and its value are embedded in a proper XML message (*VesselPositionRequest*) to invoke the service returning the position of the vessel. Similarly, position and time interval information is used to invoke a service returning the list of threats detected in a given area, including unidentified vessels. Although concepts, messages, and atoms are identified by fully-qualified names and Structural Path Model expressions, we use abbreviations in Fig. 5 to keep it readable.

VI. DATA-CENTRIC DYNAMIC INVOCATION

This section describes our mechanism for dynamic stubless invocation based on the Service Profile Model.

We exploit the expressiveness of our Service Profile Model to define an automated process for run-time message creation and analysis. The mechanism for run-time service invocation consists in (i) dynamically creating an instance of an XML request message, and (ii) sending the message to the service endpoint address over the proper transport protocol.

The structural properties for a given atom specify the structural constraints and rules that characterize the XML message part containing that atom. These properties are used to assemble/disassemble atoms in/from messages at run-time. As shown in Fig. 6, the invocation flow of a composite service is thus a sequence of operations for extracting atom values from response messages of invoked services (e.g., s_i and s_j services), and embedding them into valid XML messages for subsequent invocations (e.g., s_k). The former step is implemented through a *Document Analyzer* algorithm, as

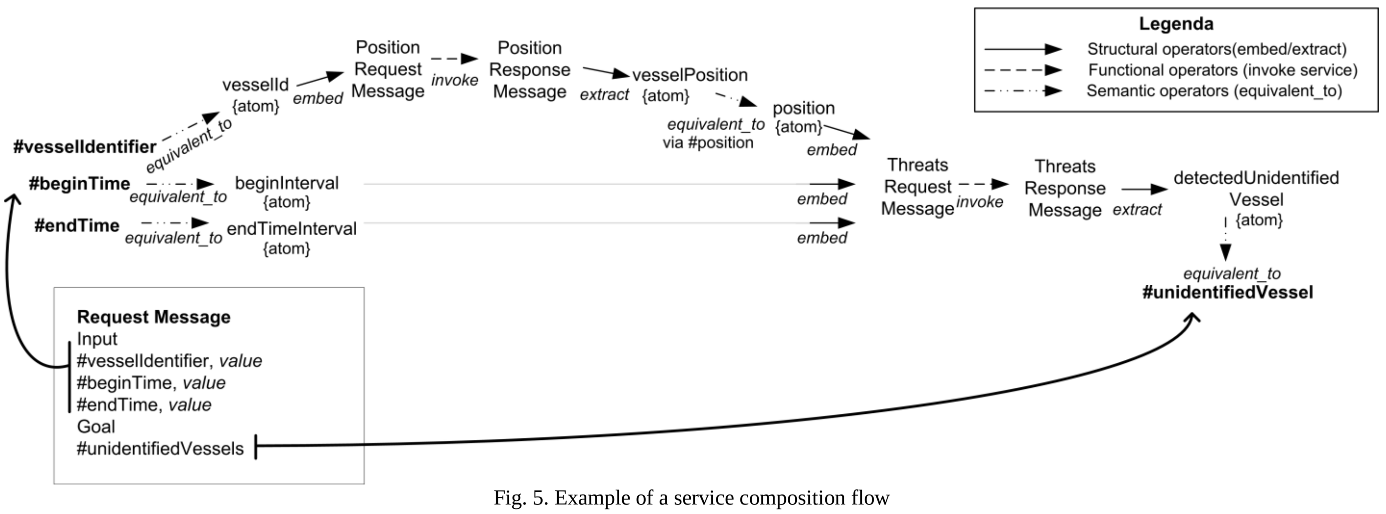


Fig. 5. Example of a service composition flow

detailed hereafter. The latter step is based on a *Document Builder* algorithm.

The *Document Builder* takes as input a list of atom structural paths and associated input data values, and gives as output an XML document whose structure complies with the structural paths and that properly embeds input data values.

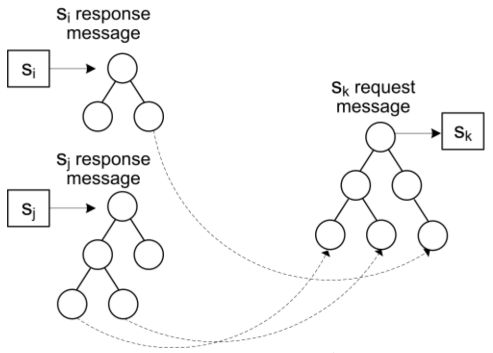


Fig. 6. Atom data extraction and insertion for dynamic invocation

Input data values are gathered from client requests or from response messages of previous invocations. The association between each input data value and the proper structural path is made by relying on equivalences implied by semantic properties. This pre-processing step is performed before invoking the Document Builder function.

For each pair (structural path, instance atom value), the algorithm first builds a string, named *instanceAtomKey*, which provides a flat textual representation of the sub-tree of the instance document containing that atom. The *instanceAtomKey* expression is based on a textual notation that is similar to the Structural Path Model. While the Structural Path Model represents structural constraints of XML Schemas (i.e., document templates), this notation represents structural constraints in instance documents (e.g., the ordering symbol represents the exact position of nodes that carry actual values in an instance document). During this step, two constraints are checked: the multiplicity of atom instances and the atom data type.

The first step of the document generation process consists in creating the root node. Then, the algorithm analyzes each *instanceAtomKey* to progressively create a subtree that is appended to the root node.

Each subtree is created by analyzing the *instanceAtomKey* string from left to right (i.e., from the root up to the leaf nodes). For each node name, the algorithm calls a *createNode* operation, which creates the node (if it does not already exist) and adds it to the parent node. The operation is invoked for each extracted node name up to a leaf node, which finally embeds the target value. Fig. 7 shows how this approach can be applied to some example input data.

The *Document Analyzer* takes as input an XML document (i.e., the response message obtained by a service invocation) and the structural paths for the atoms of that message type maintained in the Service Registry. It returns a set of pairs (value, structural path) that bind embedded response data values to the corresponding atom data structural paths. For each structural path, the algorithm implements a recursive search in the XML document from the root to the leaf node that embeds a target value. The given value is then extracted and associated to the structural path. The rest of our Document Analyzer algorithm exploits structural path information in a way similar to an XPath traversal on the document.

VII. Prototype

In order to validate the effectiveness of our Service Profile Model in enabling dynamic service composition and invocation in a working environment, we developed a prototype that serves as a proof of concept. This prototype exploits and extends the capabilities provided by a middleware infrastructure, the Service and Application Integration (SAI) system, developed in our research laboratory [33][34]. SAI is written in Java and implements many SOA principles and design patterns. It was conceived as a set of components that can be configured, assembled, and extended in different deployment configurations. It enables message exchanges across environments characterized by managerial and technological heterogeneity. The current implementation of the messaging infrastructure is powered by ActiveMQ [35], one of the leading open-source implementations of the Java Message Service specification (JMS) [36].

Fig. 8 shows the reference architecture for our prototype. The Service Registry is the SAI component that stores the profiles of registered services and exposes a set of APIs for service registration and lookup. In the registration phase, the

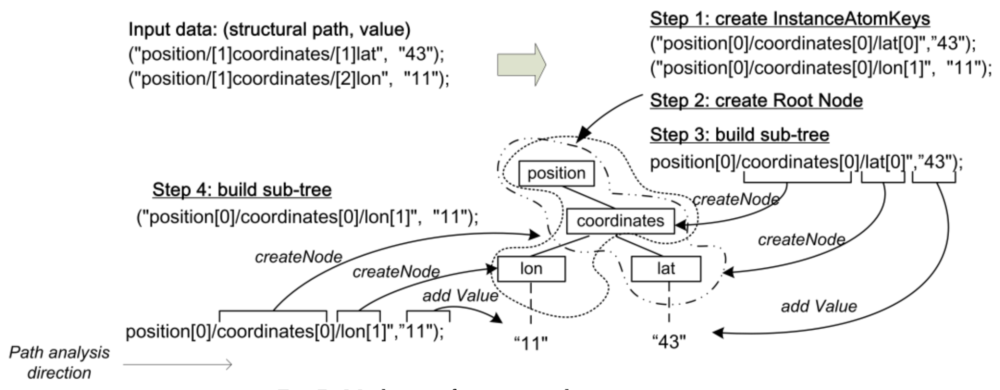

Input data: (structural path, value)
("position/[1]coordinates/[1]lat", "43");
("position/[1]coordinates/[2]lon", "11");
Step 1: create InstanceAtomKeys
("position[0]/coordinates[0]/lat[0]","43");
("position[0]/coordinates[0]/lon[1]", "11");
Step 2: create Root Node
Step 3: build sub-tree
position[0]/coordinates[0]/lat[0]","43");
Step 4: build sub-tree
("position[0]/coordinates[0]/lon[1]", "11");
position
coordinates
lon
lat
createNode
createNode
createNode
createNode
add Value
add Value
"11"
"43"
position[0]/coordinates[0]/lon[1]","11");
Path analysis direction

Fig. 7. Mechanism for run-time document creation.

Service Registry parses the service interface descriptions to generate an internal representation according to the Service Profile Model. At present, the system can interpret service descriptions written in WSDL and XML Schema. Semantic annotations for data atoms can be specified through SAWSDL annotations or SAI-specific XML files. The logic for WSDL and XML Schema parsing and the generation of structural properties are based on the XML Schema Object Model (XSOM) [37], which is the only general-purpose Java schema parser available to date (to the best of our knowledge). Registration is not restricted solely to services whose interface complies with the WSDL specification. External systems may also be accessed through customized Adaptor components that expose message-based interfaces [33].

Our Service Profile Model is expressed using the Web

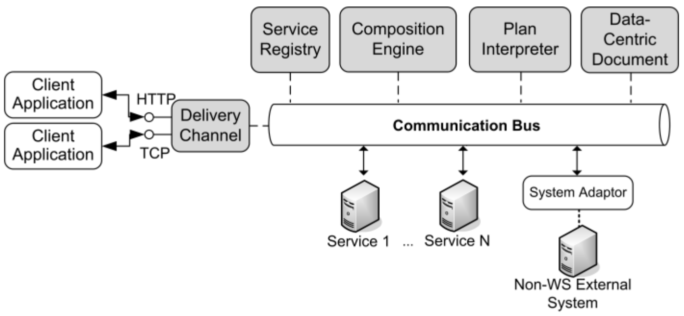


Fig. 8. Proof of concept

Ontology Language (OWL) [38]. We chose OWL because it is an expressive language with well-defined semantics and it is a W3C Recommendation. Registered service profiles are maintained in a knowledge base that was implemented using JENA, an open-source semantic framework in Java [39]. We adopted the Jena TDB triple-store storage system because it greatly facilitates the persistence of functional profiles.

The *Composition Engine* includes a STRIPS model builder that parses the functional profiles maintained in the SAI Service Registry for creating the corresponding STRIPS operators. The planner is based on a refactoring of PL-PLAN, an open-source Java library that implements the Graphplan algorithm [40]. Our extensions add support for caching computed plans and speed up client request handling by quickly identifying known-unsolvable planning problems. This component can easily be extended to integrate other planners.

The *Plan Interpreter* translates the plans produced by the Composition Engine into executable actions. More specifically, the Plan Interpreter offers two main features. First, *Endpoint Selection* implements a QoS optimization algorithm for selecting endpoints while maximizing an objective function depending on a set of quality attributes, as described in [41]. Second, *Plan Interpretation* translates the operators of the STRIPS model into executable actions according to the adopted service invocation strategy.

Dynamic stubless service invocation on a specific target endpoint is handled by two components. The *Data-Centric Document* component implements the Document Builder and Document Analyzer algorithms, based on the Java-based Document Object Model for XML (JDOM) [42]. The *Delivery Channel* offers a uniform and general-purpose message-handling interface for handling client interactions with the SAI system. The Delivery Channel can be configured by adding and removing pre- and post-processing interceptors. At present, the Delivery Channel supports request-response and one-way communication over the HTTP and JMS protocols.

The capabilities offered by the Data-centric Document and Delivery Channel components can be exploited to invoke services at run-time.

Analogously, this mechanism can be exploited to execute composite services by leveraging a workflow engine. We chose to use the jBPM process engine [43] rather than other available open-source solutions (e.g., JOpera [44] or Enhydra Shark [45]) because it is well-documented and stable. In order to automate the actions defined in the composition plan, we defined four custom jBPM activities, one for each type of STRIPS operator: *Embed Activity*, *Extract Activity*, *Equivalence Activity*, and *Invoke Activity*.

VIII. Experiments

In this section, we present the results of the experiments that we carried out to evaluate the performance of our composition and invocation mechanisms.

Our testing environment included the JUnit 4.7 testing framework and the Eclipse 3.6 Helios development environment. Tests were run on a PC equipped with an Intel Core 2 Duo processor (2.4 GHz) and 4GB DDR2 RAM.

First, we defined a set of test cases to measure the computational time needed by the dynamic composition mechanism to handle a composition request. These test cases were defined by varying the number of registered services and the number of associated semantic concepts, as shown in Table I. We defined three types of request (see Table II). Each request type has an expected solution with a given number of services and depth (i.e., the number of levels, where each level contains one or more services that can be invoked independently). For example, Fig. 9 shows the expected type of solution for request s1. Test cases were populated with a set of services designed ad hoc for matching the expected solutions, and a second set of non-matching services further populating the Service Registry.

TABLE I. Test Cases

Service Registry Population (number of registered services)	**Number of Concepts**
20	200
40	400
70	600
100	800

TABLE II. COMPOSITION REQUESTS

Request Type	# Services	# Levels
s1	4	3
s2	7	5
s3	10	7

The results of our tests are presented in Fig. 10. Our measurements illustrate the polynomial complexity of the Graphplan algorithm in the number of registered services. This was expected because the time complexity of the planning graph creation is known to be polynomial in the number of propositions and actions [30].

For the dynamic invocation mechanism, we estimated the time needed for building XML documents at run-time. This action is performed by an instance of the Data-Centric Document component. Input data are provided and processed, then the document creation is triggered. We adopted two performance metrics. First, the *Pre-processing Time* is the time needed to process the input data values and structural path expressions, in order to build a suitable representation of structural constraints for the target instance document (i.e., the *instanceAtomKey* expression mentioned in subsection VI-A). This step can be compared to the instantiation of an in-memory object representation of XML Schema types in most common service invocation frameworks (e.g., Apache WSIF, Apache Axis 2, and Java API for XML-based Web Services). Second, the *Document Building Time* is the time needed for actually creating the XML document, based on a JDOM representation. It may be viewed as a serialization time, i.e., the time needed to convert an in-memory object into an XML stream in most common service invocation frameworks (such as the ones mentioned above).

We performed several test iterations by varying the type of input data, to observe the behavior of the system when the message size increased. This increase in the message size was steered by providing an increasing number of input atom instances. We also varied the message tree-based structure by changing the nesting depth of the document tree. Figs. 11 and 12 depict the results that we collected. Both operations show an average time complexity of O(n log n).

IX. RELATED WORK

In this section, we discuss related work in the areas of dynamic service selection, composition, and invocation.

A. Service Composition and Selection

Several works on dynamic service composition exploit classic planning techniques from artificial intelligence [4]. As proposed by McIlraith et al. [46], planning-based solutions can be classified by the type of actions in such systems: (i) world-altering actions that change the state of the world, or (ii) information-providing actions that change the agent's state of knowledge.

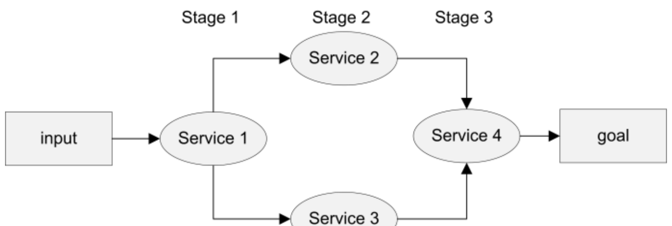


Fig. 9. Expected solution for request s1

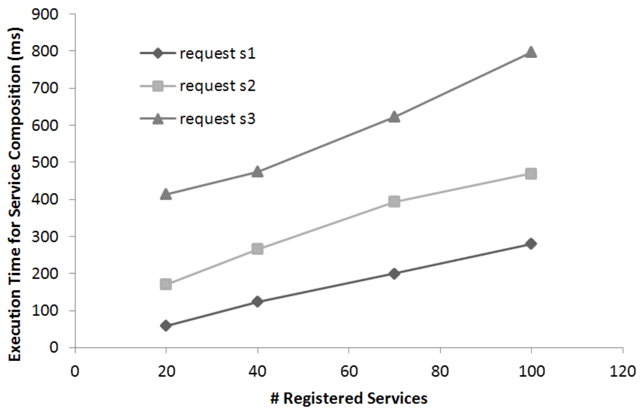


Fig. 10. Execution time for service composition

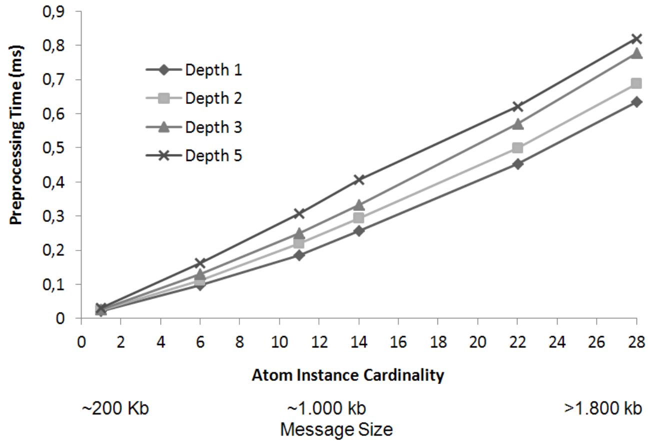


Fig. 11. Execution times for the pre-processing step

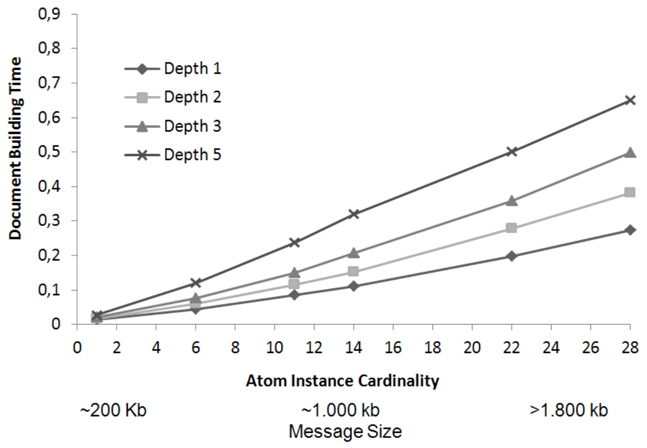


Fig. 12. Execution times for the document building step

In this paper, we focus on the approaches dealing with information-providing services. Zheng and Yan [22] model the composition of services as a syntactic matching problem, where the output parameters of a Web Service can be used as the input parameters of another Web Service. They exploit the planning model and a backward search algorithm for generating final solutions. They also enhance these classical approaches with four strategies for pruning redundant Web Services. Hewett et al. [23] compose services based on a state-space search model. A state represents a set of input parameters for services that can be added to the composition under construction. Each transition from one state to another represents an invokable service. They apply a depth-first search technique to find a solution path from an initial state to the goal state, and then a mechanism for pruning unnecessary Web Services.

Our approach is similar to these two approaches in so far as we construct a planning graph to solve an input/output matching problem (i.e., a chain of services is built by using the output parameters of a service as the input parameters of another service). While the above-mentioned works only rely on syntactic matching, our approach exploits structural and semantic matching as well. More specifically, Zheng and Yan [22] and Hewett et al. [23] assume that the messages are defined as a sequence of simple type elements, i.e., messages are defined in the WSDL description file (in the WSDL <message> element) as a collection of WSDL <part> elements declared as XML Schema simple types. Such an assumption makes it difficult to process real-world services whose interfaces may be described using different WSDL styles [47]. For instance, the message content may also be defined in the <types> element section of the WSDL document, usually by means of XML Schema constructs. This style allows for a more flexible specification of the message structure (e.g., complex types). Our Service Profile Model uses the Structural Path Model to represent these XML Schema constructs through structural operators, which are then exploited in the matching problem.

Planning-based approaches have been enhanced with semantic-based service specifications to enable semantic-based matchmaking and reasoning tasks on service contracts. Many people adopted rich semantic services models, especially OWL-S [28] and the Web Service Modeling Ontology (WSMO) [27]. Examples of approaches adopting OWL-S specifications and planning algorithms are presented by Akkiraju et al. [48] and Agarwal et al. [49]. Works based on WSMO for service discovery, composition, and invocation include the Web Service eXecution Environment (WSMX) [50] and IRS-III [51].

Both OWL-S and WSMO specify a top-down approach to semantic Web Services. They assume that a service designer first models the semantics of services, and then specifies grounding information. Kopecky and Vitvar [52] showed that this type of approach is not easily applicable to enterprise scenarios where many services are available. In order to cope with these issues, bottom-up approaches for semantic Web Services are gaining increasing interest. In this direction, the World Wide Web Consortium published the Semantic Annotations for WSDL and XML Schema Recommendation (SAWSDL) [29]. More recently, WSMO-Lite was proposed as a lightweight service ontology that defines semantic Web Service descriptions; it was published as a formal request to W3C for discussion [53]. WSMO-Lite specification activities have been mainly carried out within the SOA4All European Project [54]. Moreover, within that project, lightweight semantic service specifications were adopted for the design and implementation of iServe, a registry platform for publishing semantic annotations of services of different types (e.g., REST or Web Services APIs) with annotations in different formalisms (e.g., OWL-S or WSMO-Lite) [55]. This approach is based on a common model for service description, the Minimal Service Model (MSM), which is a simple RDF(S) ontology providing a minimal and conceptual model that captures the semantics of WSDL and RESTful services.

While our prototype implementation relies on WSDL and SAWSDL specifications, our approach can in principle support different description formalisms, thanks to our Service Profile Model. This model is similar to the Minimal Service Model proposed by Pedrinaci et al. [55] in that it represents a minimal common abstraction for service interface models.

Pedrinaci et al. [56] also exploited the Minimal Service Model to build service matchmaking techniques. Service matchmaking techniques encompass two steps: i) matching a given service request with the description of registered services according to one or more metrics; and ii) ranking the services according to the measured degree of semantic relevance [57].

Several matchmaking algorithms have been proposed to support SAWSDL-based service discovery. Most approaches calculate different similarity metrics and aggregate available measures into an overall similarity value used for service ranking. SAWSDL-MX2 [58] computes three types of matching: logical, text, and structural similarity. It adopts a Support Vector Machine (SVM) for the optimally weighted aggregation of these different matching metrics. URBE [59] calculates text and structural similarity values, and then uses a weighted aggregation scheme for service ranking. SAWSDL iMatcher [60] supports syntactic and semantic matching, as well as statistical models for aggregating different matching measures. Analogously to these SAWSDL matchmakers, our solution adopts an input/output model for describing services, rather than a full (Input, Output, Preconditions, and Effects) profile. We thus sacrifice expressivity in order to reduce the costs required for rich semantic service descriptions. The objective of a service matchmaker is to return a ranking list of relevant services to the requestor, and eventually information that allows direct interaction with providers [57]. It does not handle the composition and execution of services, as our solution does.

B. *Service Invocation*

Dynamic service invocation can be defined as the capability of a system to bind to a service and invoke one of its offered operations at run-time [13]. In practice, service invocation is usually performed by making reference to a stub (i.e., a local proxy that offers a local interface of the remote service).

In the Semantic Web research community, dynamic invocation has been often treated as a problem of mediation across client and service APIs by reconciling heterogeneities among service signatures. Semantic Web technologies were applied by Nagano et al. [61] and Lin et al. [62] to identify similarities across service signatures and build translation layers among similar service interfaces. Although they reduce the need for reprogramming client interfaces, these semantic-based approaches do require a pre-processing step of programmatic binding to a reference service interface for generating a stub. Conversely, our approach implements a generic, extensible, and truly stubless invocation mechanism that is steered at run-time by structural and semantic properties of the target services.

Several tools provide capabilities for late binding to Web Services interfaces (e.g., Apache WSIF [8], Apache Axis 2 [9], Codehaus XFire [10], Apache CXF [11], and Java API for XML-based Web Services [12]). These solutions typically support dynamic invocation by means of dynamic proxying capabilities (e.g., the *DynamicInvoker* in WSIF and the *javax.xml.ws.Dispatch* client in JAX-WS). However, as argued by Buhler et al. [13], these toolkits "*are incapable of handling complex types returned from the invoked service. This limitation is due to the fact that the returned data must be unmarshalled from the SOAP message, which in Java is not possible without having a compatible class that implements the serializable interface*". This means that the client code must include an internal representation of XML data types that has to be generated at design time in order to meaningfully exploit the information returned upon service invocation. For instance, Java-to-XML binding libraries, such as the Java Architecture for XML Binding (JAXB) [63] and XMLBeans [64], can be exploited to this purpose. More specifically, Buhler et al. [13] pointed out the following weaknesses in existing solutions: i) dynamic invocation is supported only for Web Services whose message structure does not include complex data types; ii) the handling of complex data types requires a preprocessing step that generates an internal representation of XML data in the specific programming language (e.g., Java classes); and iii) the client code is highly dependent upon specific toolkit APIs.

Buhler et al. [13] solved the latter problem by proposing a Composite Pattern for Web Service Invocation that combines two design patterns (the Bridge and Factory Method patterns [65]) for decoupling the service client code from peculiarities of specific concrete service interfaces and service invocation technologies. Leitner et al. [5] focused on the first problem by proposing the Dynamic and Asynchronous Invocation of Services Framework (Daios). Daios is a message-based service framework that allows clients to invoke remote services through a message-based stubless interface. The client request is handled by the Daios framework: Daios chooses to invoke the service interface whose input message has the lowest structural distance metric to the provided data; the framework then converts the client request data into the encoding expected by the service (e.g., a SOAP message), and launches the invocation using a proper service stack. However, Leitner et al. do not provide details about service registration and discovery in Daios, and both approaches [5][13] require a preprocessing step to compile the target service description into compatible Java classes. To this end, Leitner et al. adopted the XMLBeans library, and Buhler et al. the Java Record Object Model (JROM) [66]. Conversely, our work implements a message-oriented invocation library that allows clients as well as brokers to perform dynamic stubless invocations. Structural path expressions enable a generic mechanism for the run-time generation of XML messages that contain simple and/or complex type elements. The proposed dynamic invocation library is not strictly bounded to SOAP or to HTTP. This generic mechanism for message building and analysis can be instructed at run-time to bind to a service interface, without generating data type representations in specific programming languages.

X. Conclusion

In this work, we have proposed a lightweight and data-centric approach for achieving loosely coupled interactions among Web Service providers and consumers, via dynamic composition of services and automatic run-time binding to service endpoints. Our approach relies on a Service Profile Model that represents common aspects of service descriptions. In this model, leaf nodes in XML documents are considered first-class entities, as they contain data values in instance messages. We showed how, by means of semantic and structural properties that can be extracted at run-time from XML schema, WSDL, and SAWSDL description files, mechanisms for run-time service composition and stubless invocation have been successfully built into a prototype.

In the future, it would be useful to extend the proposed approach with ontology mediation techniques that support matching and transformation operations between XML elements and semantic concepts, or concept hierarchies (e.g., by lifting and lowering schema mappings in SAWSDL specifications). This would alleviate the need for fine-grained annotations of message and type definitions in service interface descriptions. Another direction for future work would be to extend the proposed approach to support REST-style invocations and lightweight semantic annotations (e.g., Semantic Annotations for REST [67] and MicroWSMO [68]) and to adopt data formats other than XML (e.g., JSON).

References

[1] T. Erl, *SOA: Principles of Service Design*, Prentice Hall, 2008.

[2] OASIS, Web Services Business Process Execution Language Version 2.0, OASIS Standard 11 April 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[3] OMG, Business Process Model and Notation (BPMN) Version 2.0, January 2011, http://www.omg.org/spec/BPMN/2.0/

[4] J. Rao and X. Su. "A Survey of Automated Web Service Composition Methods", in Proc. of First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), San Diego, California, USA, July 2004.

[5] P. Leitner, F. Rosenberg, and S. Dustdar, "Daios: Efficient Dynamic Web Service Invocation", *IEEE Internet Computing*, vol. 13, no. 3, May 2009, pp. 72-80.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL"), W3C Note, 15 March 2001, http://www.w3.org/TR/wsdl.

[7] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation, 26 November 2008, http://www.w3.org/TR/xml/.

[8] Apache Web Services Invocation Framework (WSIF), http://ws.apache.org/wsif.

[9] Apache Axis 2, http://ws.apache.org/axis2.

[10] Codehaus XFire, http://xfire.codehaus.org.

[11] Apache CXF, http://cxf.apache.org.

[12] Java API for XML-Based Web Services (JAX-WS) 2.0, JSR 224, May 2006, http://jcp.org/en/jsr/detail?id=224.

[13] P. Buhler, C. Starr, W. H. Schroder, and J.M. Vidal, "Preparing for Service-Oriented Computing: A Composite Design Pattern for Stubless Web Service Invocation", in Proc. of 2005 IASTED Conf. on Software Engineering (SE 2005), Innsbruck, Austria, February 2005, pp.276-281.

[14] J. Kramer, J. Magee, and M. Sloman, "A software architecture for distributed computer control systems", *Automatica*, vol. 20, no. 1, 1984, pp. 93-102.

[15] E. Wilde and R.J. Glushko, "Document Design Matters", *Communications of the ACM*, vol. 51, no. 10, pp. 43-49, 2008.

[16] W3C, SOAP Version 1.2 Part 0: Primer (Second Edition) W3C Recommendation, 27 April 2007, http://www.w3.org/TR/soap12-part0/

[17] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, IETF, July 2006.

[18] D.C. Fallside and P. Walmsley, XML schema part 0: Primer, Second edition (2004), W3C Recommendation, 28 October 2004.

[19] M. Granitzer, V. Sabol, K.W. Onn, D. Lukose, and K. Tochtermann, "Ontology Alignment—A Survey with Focus on Visually Supported Semi-Automatic Techniques", *Future Internet*, vol. 2, no. 3, pp. 238-258, 2010.

[20] F. Lécué, S. Salibi, P. Bron, and A. Moreau, "Semantic and Syntactic Data Flow in Web Service Composition", in Proc. of 2008 IEEE International Conference on Web Services (ICWS 2008), Beijing, China, September 2008, pp. 211-218.

[21] P. Walmsley, "The Importance of Schema Design in SOA*", SOA Magazine*, no. XXXVII, March 2010, http://www.servicetechmag.com/I37/0310-1

[22] X. Zheng and Y. Yan, "An Efficient Syntactic Web Service Composition Algorithm Based on the Planning Graph Model", in Proc. of 2008 IEEE International Conference on Web Services (ICWS 2008), Beijing, China, September 2008, pp. 691-699.

[23] R. Hewett, P. Kijsanayothin, and B. Nguyen, "Scalable Optimized Composition of Web Services with Complexity Analysis", in Proc. of 2009 IEEE Int. Conf. on Web Services (ICWS 2009), Los Angeles, CA, USA, July 2009, pp. 389-396.

[24] J. Tekli and R. Chbeir, "A novel XML document structure comparison framework based-on sub-tree commonalities and label semantics", *Journal of Web Semantics*, vol. 11, pp. 14-40, March 2012.

[25] W3C, XML Path Language (XPath) 2.0, W3C Recommendation 23 January 2007, http://www.w3.org/TR/xpath20/.

[26] W3C, XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Recommendation 23 January 2007, http://www.w3.org/TR/xpath-datamodel/.

[27] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domingue, *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer, 2007.

[28] D. Martin, M. Burstein, D. McDermott, S. McIlraith, M. Paolucci, K. Sycara, D.L. McGuinness, E.Sirin, and N.Srinivasan, "Bringing Semantics to Web Services with OWL-S", *World Wide Web*, vol. 10, no. 3, September 2007.

[29] J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell, "SAWSDL: Semantic Annotations for WSDL and XML Schema", *IEEE Internet Computing*, vol. 11, no. 6, pp. 60-67, Nov.-Dec. 2007.

[30] A. Blum and M. Furst, "Fast Planning Through Planning Graph Analysis", *Artificial Intelligence*, vol. 90, pp. 281-300, 1997.

[31] M. Alrifai and T. Risse, "Efficient QoS-aware Web Service Composition", in 3rd Workshop on Emerging Web Services Technology (WEWST 2008) in conjunction with ECOWS 2008, Dublin, Ireland, November 12, 2008.

[32] E. Kutluhan, S. N. Dana, and V. S. Subrahmanian, "When is planning decidable?", in Proc. of First Int. Conf. on Artificial Intelligence Planning Systems, College Park, MD, USA, June 15-17, 1992, pp.222-227.

[33] D. Parlanti, F. Paganelli, D. Giuli, "A Service-Oriented Approach for Network-Centric Data Integration and Its Application to Maritime Surveillance", *IEEE Systems Journal*, vol.5, no.2, pp.164-175, 2011.

[34] F. Paganelli, D. Parlanti, D. Giuli, "Message-Based Service Brokering and Dynamic Composition in the SAI Middleware", in Proc. of 2010 IEEE International Conference on Services Computing (SCC 2010), Miami, FL, USA, July 2010, pp.474-481.

[35] The Apache Software Foundation. ActiveMQ. http://activemq.apache.org/.

[36] M. Richards, R. Monson-Haefel, and D.A Chappell, *Java Message Service*, O'Reilly, 2009.

[37] XML Schema Object Model (XSOM) https://xsom.dev.java.net/

[38] Web Ontology Language, W3C recommendation, http://www.w3.org/TR/owl-guide/.

[39] Jena: Semantic web framework, http://jena.sourceforge.net/documentation.html.

[40] P. Fournier-Viger, and L. Lebel, PL-PLAN, Java Open-Source AI Planner, http://plplan.philippe-fournier-viger.com/index.html.

[41] F. Paganelli, T. Ambra, D. Parlanti, and D. Giuli, "A semantic-driven Integer Programming Approach for QoS-aware Dynamic Service Composition", in Proc. of 50th FITCE Congress ICT: Bridging an Ever Shifting Digital Divide (FITCE 2011), Palermo, Italy, August 2011.

[42] JDOM, http://www.jdom.org/.

[43] JBoss jBPM, http://www.jboss.org/jbpm.

[44] C. Pautasso, T. Heinis, G. Alonso, "JOpera: Autonomic Service Orchestration", *IEEE Data Engineering Bulletin*, vol. 29, no. 3, September 2006.

[45] Enhydra Shark, http://shark.ow2.org/doc/1.0/index.html

[46] S. McIlraith and T. C. Son, "Adapting Golog for composition of Semantic Web services", in Proc. of 8th Int. Conf. on Knowledge Representation and Reasoning (KR2002), Toulouse, France, April 2002.

[47] R. Butek, "Which style of WSDL should I use?", IBM Technical library, 2005, http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/

[48] R. Akkiraju, B. Srivastava, A.-A. Ivan, R. Goodwin, and R. Tanveer Syeda-Mahmood, "SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition", in Proc. of 2006 Int. Conf. on Web Services (ICWS 2006), Chicago, IL, USA, Sept. 2006, pp. 37-44.

[49] V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava, "Synthy: A system for end to end composition of web services", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 4, pp. 311-339, December 2005.

[50] A. Haller, E. Cimpian, A. Mocan, E. Oren, C. Bussler, "WSMX - a semantic service-oriented architecture", in Proc. of 2005 IEEE Int. Conference on Web Services (ICWS 2005), vol. 1, Orlando, FL, USA, July 2005, pp. 321- 328.

[51] J. Domingue, L. Cabral, S. Galizia, V. Tanasescu, A. Gugliotta, B. Norton, C. Pedrinaci, "IRS-III: A broker-based approach to semantic Web services", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 2, pp. 109-132, April 2008.

[52] J. Kopecky, and T. Vitvar, “WSMO-Lite: Lowering the Semantic Web Services Barrier with Modular and Light-Weight Annotations”, in Proc. of 2008 IEEE Int. Conf. on Semantic Computing (ICSC 2008), Santa Clara, CA, USA, August 2008, pp. 238-244.

[53] D. Fensel, F. Fischer, J. Kopecký, R. Krummenacher, D. Lambert, T. Vitvar, “WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web”, W3C Member Submission 23 August 2010. http://www.w3.org/Submission/WSMO-Lite/.

[54] SOA4All European Project, http://www.soa4all.eu/.

[55] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky and J. Domingue, “iServe: a linked services publishing platform”, in Proc. of the 1st Workshop on Ontology Repositories and Editors for the Semantic Web (ORES 2010), Heraklion, Greece, May 2010.

[56] C. Pedrinaci, D. Lambert, M. Maleshkova, D.Liu, J. Domingue, and R. Krummenacher, “Adaptive Service Binding with Lightweight Semantic Web Services”, in Schahram Dustdar and Fei Li (Eds.), *Service Engineering: European Research Results*, Springer, 2010.

[57] M. Klusch, “The S3 Contest: Performance Evaluation of Semantic Service Matchmakers”, in *Evaluating Semantic Web Services Advancement through Evaluation*, M.B. Blake, L. Cabral, B. Knig-Ries, U. Kster, D. Martin (Eds.), Springer, 2012.

[58] M. Klusch, P. Kapahnke, and I. Zinnikus, “Adaptive Hybrid Semantic Selection of SAWSDL Services with SAWSDL-MX2”, *International Journal on Semantic Web and Information Systems* (IJSWIS), vol. 6, no. 4, pp. 1-26.

[59] P. Plebani and B. Pernici, “URBE: Web Service Retrieval Based on Similarity Evaluation”, *IEEE Transactions on Knowledge and Data Engineering* , vol.21, no.11, pp.1629-1642, Nov. 2009.

[60] D. Wei, T. Wang, J. Wang, and A. Bernstein, “SAWSDL-iMatcher: A customizable and effective Semantic Web Service matchmaker”, *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 9, no. 4, pp. 402-417, December 2011.

[61] S. Nagano, T. Hasegawa, A. Ohsuga, and S. Honiden, “Dynamic invocation model of Web services using subsumption relations”, in Proc. of 2004 IEEE Int. Conf. on Web Services (ICWS 2004), San Diego, CA, USA, July 2004, pp. 150- 156.

[62] B. Lin, N. Gu, and Q. Li, “A Requester-based Mediation Framework for Dynamic Invocation of Web Services”, in Proc. of 2006 IEEE International Conference on Services Computing (SCC 2006), Chicago, IL, USA, September 2006, pp.445-454.

[63] Java Architecture for XML Binding (JAXB) 2.0, Java Specification Request (JSR) 222, http://jcp.org/en/jsr/detail?id=222

[64] Apache XMLBeans, http://xmlbeans.apache.org/

[65] E. Gamma, *Design patterns : elements of reusable object-oriented software*, Addison-Wesley, 1995.

[66] IBM Alphaworks. Java Record Object Model (JROM), http://www.alphaworks.ibm.com/tech/jrom, 2002.

[67] A. P. Sheth, K. Gomadam, and J. Lathem, “SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups”, *IEEE Internet Computing*, vol. 11, no. 6, 2007.

[68] J. Kopecky, and T. Vitvar, “MicroWSMO”, WSMO Working Draft, February 2008, http://www.wsmo.org/TR/d38/v0.1/20080219/.

Federica Paganelli (M’07) received a Ph.D. degree in Telematics and Information Society from the University of Florence, Italy, in 2004. She is a Senior Researcher at the National Interuniversity Consortium for Telecommunications (CNIT), Italy. Her research interests include context-aware systems, Ambient Intelligence, service-oriented computing, and next generation networks.

David Parlanti received a Ph.D. degree in Telematics and Information Society from the University of Florence, Italy, in 2007. From 2007 to 2009, he was a Researcher at CNIT, Italy. Since 2010, he is with Negentis, a System Integration company in Firenze, Italy. His research interests are focused on message-oriented systems, P2P distributed systems, and SOA/EDA architectures for systems integration.