Journal of Web Engineering, Vol. 15, No. 3&4 (2016) 310–338

A SEMANTIC APPROACH FOR DYNAMICALLY DETERMINING COMPLEX COMPOSED SERVICE BEHAVIOUR

CARLA VAIRETTI[a]
Computer Science Department. Pontifcia Universidad Catlica de Chile
Av. Vicuña Mackenna 4860, Santiago, Chile
cmvairet@uc.cl
Facultad de Ingeniería y Ciencias Aplicadas. Universidad de los Andes.
Mons Alvaro del Portillo 12445, Santiago, Chile
cvairetti@uandes.cl

ROSA ALARCON
Computer Science Department, Pontifcia Universidad Catlica de Chile
Santiago, Chile
ralarcon@ing.puc.cl

JESUS BELLIDO
Computer Science Department, Pontifcia Universidad Catlica de Chile
Santiago, Chile
jbellido@uc.cl

Received April 28, 2015
Revised January 7, 2016

Dynamic Web services composition aims to generate a composition plan at run-time. Semantic-based techniques rely on annotating services to facilitate the discovery of the service components that satisfy a user need (matchmaking). The matchmaking process places most attention on service selection rather than on the behaviour of the composed service, and the service components are arranged considering simple control-flow patterns (mainly sequence). In real life scenarios, however, composed service behaviour follows complex control-flow patterns that satisfy the needs of business processes, which are generally defined through manual service composition. In this paper we present a technique to derive complex composed service behaviour semantics, such semantics make possible to dynamically and automatically discover complex services compositions. We have implemented and tested our technique with a known dataset with better performance when compared to simple service composition strategies.

Keywords: Dynamic service composition, semantic Web services, service behaviour, control-flow
Communicated by: M. Gaedke & Rossi

1. Introduction

Web service composition is the process of combining the functionality of diverse services (components) into a new service that provides aggregated value and can be part of another composed service [17]. Service composition requires defining the order and conditions to

[a]Computer Science Department, Pontifcia Universidad Catlica de Chile

selected, bind and invoked services. These tasks can be performed automatically or manually, at design-time (static) or at run-time (dynamic). Dynamic and automatic composition is desirable because it contributes to reduce the development costs of creating new services. It can also assists developers to discover services among a myriad of existing services and to deal with the failure of a component or a whole composed service on real time, facilitating composed services to adapt to contextual changes.

A popular strategy for supporting dynamic and automatic service composition exploits service signature, that is, services input and output to determine services dependencies, deriving a composition plan that can be seen as a graph [15]. Most research focus on enriching services signature with additional information (pre and post conditions, quality, conceptual semantic models, business rules, etc.) in order to improve services dependencies. Standards such as SAWSDL [28] allow service providers to annotate web service descriptions (WSDL) with references to semantic elements without prescribing a semantic model, which is kept separated from the description. Popular semantic approaches such as OWL-S and WSMO describe service semantics reling on expressive knowledge representation formalisms such as OWL [34] and WSML [13] respectively, along with rule languages. Domain ontologies for both OWL-S and WSMO are rich and complex and the development on either platform demands significant expertise and knowledge from designers and developers on subjects such as the corresponding domain ontology, the platforms, and the tools that enable the execution of semantic Web Services. These characteristics imply an important limitation to the scalability of these approaches [39, 4], for this reason, lightweight approaches such as WSMO-Lite [14], and the Minimal Service Model (MSM) [40]. Research on semantic-based dynamic composition place a strong emphasis on the discovery of suitable candidates for a composition [10], while the behaviour of the composed service is either highly complex, over-simplified [29] or ignored, for instance, WSMO-Lite does not support a control-flow infrastructure but instead this one is provided by WSMO. Manual techniques on the other hand, allow full control on the specification of the service behaviour, resulting into a variety of complex control flows patterns that satisfy the various needs and constraints of the business processes [46, 47].

Automatic composition is a challenge that tends to become more difficult when the number of services increases, which is worsened if connections between services are complex (i.e. when complex control-flow patterns are included). Some approaches that follow artificial intelligence planning [23, 48, 26, 41, 55] derive the sequence of actions required to reach a goal state (required outputs) from a initial state (inputs and preconditions). These techniques typically work well for small repositories with a high number of constraints. Most of these proposals have some drawbacks: high complexity, high computational cost and inability to maximize the parallel execution of web services. Others [6, 18, 42] deal with a huge number of services but they do not guarantee to obtain an optimal solution, are extremely slow and memory intensive. An approach that is similar to us [43] finds a valid composition considering the matching of the input-output message at a semantic level. The approach scales better than other techniques with huge number of services, and also shows a great performance over large repositories. However, they can discover only two of the most important control-flow patterns: sequence and parallel.

In this paper, we present our approach for dynamic service composition (CompoSWS) that exploits service signature and semantic annotations along with rules to identify simple and

complex control-flow patterns between services at publishing-time (i.e. when a service provider makes its service available in our platform). Services are connected through such patterns forming a graph that is pre-calculated and represent the behavioural semantics of a potential composed service. A composed service can be dynamically and automatically discovered and assembled into an executable service at consuming-time (i.e. when an consumer requests a non existent service but whose functionality can be provided through a services subgraph). We propose also to extend the Minimal Service Model (MSM), which is a lightweight ontology that captures (part of) the semantics of both Web services and Web APIs in a common model focuses on services signature and facilitates our approachs scalability. We validate our approach theoretically through a complexity analysis and experimentally on a known dataset of 980 services, both at publishing-time and consuming-time, in terms of performance (response time), and scalability (compositions of various sizes). Our results are promising and suggest that our approach could be used in an on-line fashion. Our experience indicated some limitations of SPARQL 1.1. Specification when querying subgraphs [5] that was resolved by defining incremental queries (i.e. progressively reducing the search space).

This paper describes a technique to derive complex composed service behaviour semantics that:

- Extends the MSM ontology in order to allow the specification of simple and complex control-flow patterns based on the services signature;
- Enables the automatic discovery of such patterns through a set of rules;
- We also present the algorithms and queries required to dynamically pre-compute all the possible combinations between services taking into account service behaviour (derived from the control-flow patterns); and the algorithm and queries required to discover composed services.

The contributions of this paper are two; first we improve the performance, in terms of response time, of generating composed services without requiring in memory calculus, which may facilitate scalability of our approach through horizontal scalability. Second we allow the generation of more elaborate compositions that correspond to complex business patterns adopted in most real scenarios, without losing performance when compared to approaches that only consider simple business patterns.

2. Related Work

The most popular technique for dynamic service composition is based on service signature; it considers the dependencies between inputs and outputs in order to generate a composition plan at runtime. Plan generation uses techniques such as dynamic forward or backward search, data dependency and control-flow. The plan can be represented as a directed acyclic graph where nodes are services and arcs correspond to the dependencies between them. Such dependency is established from the service signature (Input, Output), services semantic distance, or services similarity (perfect-match, plug-in, subsume, y zero-match, etc.) [15]. The composition plan includes the services invocation control-flow. Some approaches, such as Linked-OWL [2], require that the user defines the composition plan as an abstract workflow, services are stored in a repository and each service component is searched for using a SPARQL query.

Others propose a Petri net-based algebra for modelling control-flow [21]. Service behaviour is modelled through six control-flow patterns: sequence, alternative, iteration, arbitrary sequence, parallel with communication, discriminator, selection, and refinement. OWL-S is a well-know semantic service model that includes basic control flow patterns (invocation, sequence, alternative, join and split) [56]. A UML- based universal language considering simple and complex interaction has been also proposed [29]. Workflows are considered complex interaction patterns (sequence), but the authors do not specify how such workflows could be modelled. In [50, 46] 43 control-flow patterns are presented in great detail.

Service discovery is achieved by defining a matching degree between services (matchmaking) and the client's request. Matchmaking is considered logics-based, if an ontological structure is used to determine a similarity between the services and the request; non-logical, if syntactic, structural or numeral strategies are used, or hybrid. In YASA [11] a hybrid matchmaking mechanism pre-selects a set of services, and determines a services matching degree using a IOPE (Input, Output, Preconditions, Effects) logical approach (exact match, subsumption, etc.); finally, the matching degree is weighted using non logical strategies (min-average, cupid, and combinatorial algorithms). Logical and statistical matching degree is broader in IO OWLS- MX [27] the user determines the matching degree, and the similarity threshold, and the service provider, requester and matchmaker share a minimal vocabulary with mapping rules (synonyms based on a WordNet thesaurus) to classify service requests input and output concepts. In [7], a hybrid PE (Preconditions, Effects) algorithm ranks published services according to the semantic distance of concepts (counting edges). WordNet is used to determine synonymy and concept subsumption. SAM is an IO matchmaking algorithm for OWL-S that considers semantic descriptions for requested and provided services [10]. The algorithm simplifies services into trees and creates a BF-hypergraph representing the dependencies among matched services; the dependency graph in analysed to determine whether it satisfies the user request or additional input requests are required from the client to produce a full match.

Semantic-based service composition techniques rely on semantically annotated services to facilitate service discovery. Annotations make explicit the semantics of the input and output parameters, as well as the service goals among others. For instance in Kill and Nam [24], conceptual relationships between services parameters are used to find the semantically close services that satisfy the users requirement. This technique is based on model checking and a matchmaking process. Other solutions are based on various description languages capturing different services semantics. For instance, OWL- S proposes three ontologies specifying a service (a service profile indicating the service goals, limitations, quality and requirements for the service consumer), usage (a service model), and access (service grounding). A service model component, the Process, describes properties such as inputs, outputs, preconditions, parameters, and effects; the Process Control component, describes the processing state and together they allow designers to create workflows. Unlike OWL-S, the Web Service Modeling Ontology (WSMO) includes an execution framework and a set of four ontologies that describe the information used by the other ontologies; the objectives fulfilled when executing the service; the services capabilities and signature; and the mapping between components. Service discovery is a three-step process involving the service signature and goal, whereas service composition is generally addressed through forward-chaining techniques that determine the set

of valid state transitions (i.e. service invocation) in order to achieve a goal [16]. WSMO-Lite service ontology [53] is an extended Web service specification stack, adding semantic layers that offer richer descriptions for Web services with the goal of the maximal compliance with Web standards. SAWSDL (Semantic Annotations for WSDL) is a W3C recommendation for semantic service descriptions, which extends XML-based WSDL with semantic annotations without imposing a representation language (e.g. RDF, OWL, WSML, etc.), and without prescribing a service semantic model (i. e. could be compatible with OWL-S, WSMO or other models). Some WSDL elements can be annotated with a modelReference attribute that refers to the equivalent concept in some semantic model through a URI.

On other hand, the DSD [25] language describes services from a pure state-based approach; it requires services to declare its effects and pre-conditions. Services domain is modelled through a hierarchical ontology specialized in various layers down-to instances. Instance sets (instances subgraphs with constrained attributes) serve as a medium to specify unambiguously consumers request and providers capabilities. A service is composed dynamically from a request. In DSD, and such request is an instance set that specifies the expected state of the world after a successful service execution. For YASA, a query-formatted document (i.e. an extended SAWSDL description with annotations on the interface, operation, input and output elements) representing an abstract description of the expected service is used as the request specification. For WSMO, the desired goals as well as the input values are specified [16] through abstract goal templates describing functional capabilities and constraints that are instanced and customized by users.

Services orchestration aims to generate a composition plan determining the service components, the data to be interchanged and the control-flow regulating services interaction. Service orchestration is the most popular paradigm in REST service composition research. For instance, the JOpera framework [36] proposes a visual language and an execution platform for building large applications including multiple REST services. In JOpera, the orchestrator is implemented as a central (composite) resource that drives the control and data flow. Both flows are visually modeled as two separate design documents producing a BPEL compatible executable program for orchestration engines. Other works such as a BPEL extension for REST [37] and a BPEL-inspired workflow composition language called Bite [45] are used to describe control/data flow and data transformations for web service composition and control flow dependencies are modeled and implemented using a Petri Net in [3]. A set of control-flow patterns that implement stateless REST service composition are described in [8]. Authors describe a technique for decentralized REST services composition that takes into account the constraints of REST architectural style in the composition process. The implemented control flows follow a choreography paradigm implemented through callbacks and redirections.

Semantic REST service composition approaches model the composition as graph patterns [30]. For instance, Verborgh et al. propose an RDF-based approach for describing RESTful services where a service composition is implemented through SPARQL queries (N3). The control-flow is modeled as query patterns following the RESTdesc language; data flow is dynamically resolved when the query is performed and the served representations can be later processed [52]. Mismatches between data formats, fully supported in JOpera, are not considered [51]. Semantic Web technologies are used to model contextual information from users, sensors and things so that machine-clients can make sense from the responses [22].

Other approaches for REST service composition focus on service description. Proposals include WADL [20], WSDL 2.0 [12], and SA-REST [31]. These descriptions facilitate the automation of machine-client and RESTful services interaction. These languages are strongly influenced by existing imperative service description languages (input/output) and do not capture well the resource-centric nature of RESTful WSs (transitions of resources). ReLL [3] differs from the other three in that it is hypermedia-centric, supports REST architectural constraints requiring less coupling between clients and services. Semantic REST service descriptions have been also proposed. For instance, hRESTS [33] proposes a microformat to annotate HTML service descriptions that can be used also by crawlers and search engines to find services. The microformat extends the HTML description with semantic annotations so that RESTful services can be discovered, composed and invoked automatically. Four aspects of service semantics: information model, functional semantics, behavioral semantics and nonfunctional descriptions, instances are modeled by MicroWSMO [33]. SWEET [33] supports users in searching for suitable domain ontologies and in making semantic annotations in MicroWSMO in order to provide a higher level of automation on tasks with RESTful services, such as discovery and composition. WSMO-Lite [54] ontology is used for describing the content of semantic annotations in WSDL.

3. Composing Web services considering complex control-flow patterns

Web service composition requires determining the service components as well as the order in which services are invoked. Such choices can be made dynamically and automatically at consuming-time (i.e. when a consumer requests a non existing service) by examining the characteristics of a set of known services. As described before, services signature can be used to determine both service components and the dependencies among them. Typically such dependencies are simple sequence and alternative control-flow patterns (e.g. consume service A first in order to produce and output that serves as an input for the subsequent service B).

Composed services in the real world, however, follow complex control-flow patterns in order to fulfil the requirements and constraints of real world business processes [49]. Furthermore, business process modelling comprehends up to 43 well-known control-flow patterns [50, 46]. Semantic Web service composition, on the other hand, considers various properties to determine a composed service, however, the few service model ontologies (OWL-S) that contain elements that make possible to produce complex control-flow patterns are extremely complex and verbose and control-flow related concepts and relationships cannot be derived automatically but have to be included in the model manually, at design-time.

In order to face such problem, in this paper we extend a well-known and simple semantic Web service ontology (MSM, the minimal service model), with minimal concepts and relationships that make possible to represent relationships among services corresponding to complex control-flow patterns. In this way it is possible to discover composed services as subgraphs where services are interlinked following complex control-flow patterns.

In this section, we present a real-world business process model that includes various control-flow patterns as a motivating example (subsection 3.1). Then, we extend the MSM ontology to support complex control-flow patterns (subsection 3.2) and then we present our approach to derive 6 control-flow patterns and the corresponding semantic relationships (subsection 3.3).

3.1. *Motivating and example: Finding a service to apply for a travel reimbursement*

We introduce a business case scenario that is used along the paper to illustrate our approach. It is inspired on the University of Minnesota travel reimbursement process (*https* : *//www*1.*umn.edu/ohr/pay/reimbursements/index.html*). For demonstrating the effect of all the composition patterns we have added complexity to the final step (12) of the process. Figure 1 presents a business process model for the business case; it comprehends several steps that we summarize as follows:

1. An employee must retain detailed itemized receipts for expenses of $25 or more, excluding meals, and s/he must prepare an Employee Expense Worksheet.
2. The employee must sign the worksheet.
3. The employee must attach the receipts to the worksheet and for each receipt, the system must validate if the costs are within the margins accepted by the university.
4. After the worksheet is completed, it is sent to a Preparer and s/he verifies that the expenses meet the University (and/or applicable sponsored fund) policy and procedures.
5. The Preparer ensures that receipts are included as required and asks the employee for any missing receipts.
6. If any rates claimed for applicable charges (hotel, mileage, per diem, etc.) exceed the University limits, the Preparer contacts the employee, and informs him/her of any adjustments made to the total reimbursement.
7. The Preparer prints a barcoded Expense Report from the financial system after submitting it for approval.
8. The Preparer also attaches the worksheet, receipts, and other support documentation to the printed Expense Report and forwards it to the Approver(s).
9. The Approver reviews the Employee Expense Worksheet to verify if inadequate substantiation exists for any expense item.
10. If there is an inadequate substantiation, the Approver must request the appropriate substantiation for the items in question. In addition, the Approver will deny any unsubstantiated expense reimbursement if it is not accompanied by an appropriate substantiation.
11. The Approver may choose to deny any reimbursement request not submitted within the established timeline.
12. Once the Employee Expense Worksheet and attached information is appropriate, s/he approves the transaction and determines the characteristics of the reimbursement such as the sponsored funds from where the money must be transferred, the payment mode (bank, cash or Paypal) and the number of payment installments.

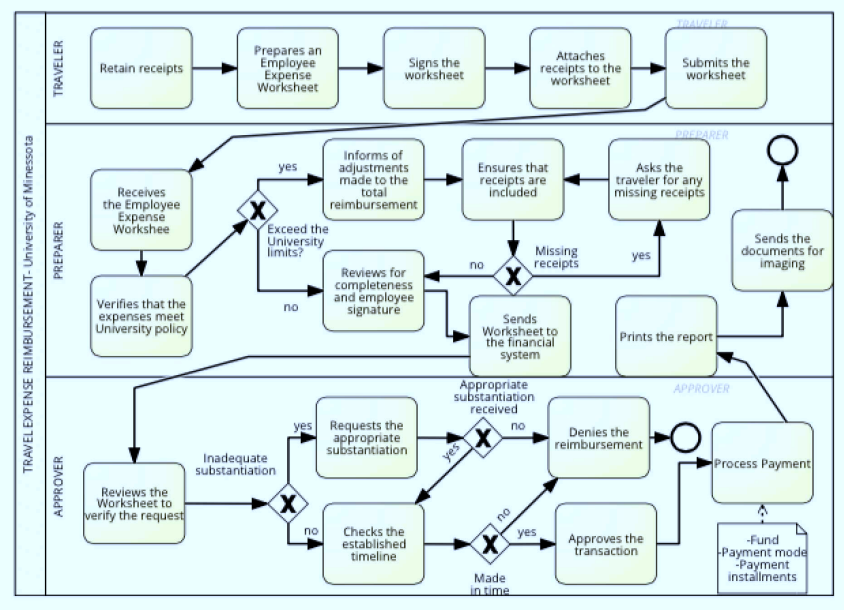


Fig. 1. PMN model of a travel expense reimbursement process based on the Minnesota University reimbursement process.

Let's suppose that even though various services provide a partial solution for the problem, a composed service providing the whole functionality is not yet available. In this scenario, users (e.g. a business specialist, a software engineer, an IT analyst, etc.) issue a composition request, which in our approach is an XML file indicating the desired characteristics of a service, as seen in Figure 2. The request can be determined through dialogs as the composition is built, but since our focus is the composition itself, we will let out this feature and will assume that the service request contains all the required information.

Let's assume that a user issues the service request as described in Figure 2. In this request, a goal element (Figure 2, line 4) is used to describe the desired activity that an atomic or composed service will provide (e.g. to obtain a ProcessedReimbursement). The parameters element (line 6) describes the input and output information that the user requesting the service is providing. Note that we refer to the concept (semantics) associated to such parameters, instead of considering it a data type or a value since the latter will be provided at runtime. In addition, our algorithm requires at most one output but zero or more input concepts (Figure 2, lines 6 to 21). Additional constraints may be provided, for instance the user chooses the sponsored fund that allows maximum approval limits (Figure 2, line 23), to reimburse through a bank account (payment mode: bank, cash or paypal) (line 24) and the number of payment installments, in this case the request specifies only three installments (line 25). These expressions follow an XPath notation, they are resolved dynamically and bound to the appropriate control-flow pattern depending on the concepts they refer to (i.e. input, output or goal).

As we can see in Figure 1 and the request issued by the client (Figure 2), a model (e.g. an ontology) that supports real world service composition must be able to represent complex

```
<?xml xmlns:sb="http://soc.ing.puc.cl/CompoWS/ServiceBehavior"
 xmlns:reimbursement="http://www.university.org/finance/reimburse.owl"
 targetNamespace="http://example.com/requestReimbursement.xls"
 xmlns:tns="http://example.com/requestReimbursement.xls"
 xmlns:rq="http://www.soc.ing.puc.cl/CompoWS/request"
 version="1.0" encoding="UTF-8"?>
 <xsd:complexType name="Persona">
            <xsd:sequence>
                <xsd:element name="names" type="xsd:string" />
                <xsd:element name="surnames" type="xsd:string"/>
                <xsd:element name="dateBirth" type="xsd:date"/>
                <xsd:element name="personalAccount" type="xsd:integer"/>
            </xsd:sequence>
 </xsd:complexType>
<rq:request>
 <sb:Goal> <!-- Describes the goal to be achieved -->
        <rq:modelref>reimbursement:ProcessedReimbursement </rq:modelref>
 </sb:Goal>
 <rq:parameters><!-- Describe inputs and the output to be obtained -->
        <rq:paramIn> <!-- Inputs can be more than one -->
            <rq:name>receipts</name>
            <rq:modelref>reimbursement:Receipts</rq:modelref>
            <rq:value>[id2014,id2023,id2314,id2456]</rq:value>
        </rq:paramIn>
        <rq:paramIn> <!-- Inputs can be more than one -->
            <rq:name>person</name>
            <rq:modelref> reimbursement:PersonalData</rq:modelref>
            <rq:value type="tns:Persona">P</rq:value>
        </rq:paramIn>
        <rq:paramOut>
            <rq:name>result</name>
            <rq:modelref>reimbursement:ReimbursementResult </rq:modelref>
        </rq:paramOut>
 </rq:parameters>
 <rq:guard> <!-- Used to contain the service behaviour of composite. XPath expressions -->
  <rq:expression> math:max(reimbursement:FundApprovalLimit) </expression>
  <rq:expression> contains:(reimbursement:ReimburseByBankAccount") </expression>
  <rq:expression> for-each(reimbursement:NumPaymentInstallments,3) </expression>
 </rq:guard>
</request>
```

Fig. 2. The user request as specified in an XML document.

control-flow patterns such as those arising in the example (alternative service selection, parallel invocation, and various synchronization patterns), as well as certain constraints (conditional selection of responses, and iteration) that affect or result in additional control-flow patterns (e.g. iteration). Existing Web services ontologies that consider control-flow do not consider complex control-flow (such as iteration or conditional selection of responses) and do not provide extensibility elements to model such new patterns easily.

3.2. *Extending MSM to support complex control-flow patterns*

In order to support complex control-flow patterns, we propose a simple extension to the MSM service ontology (Figure 1). In MSM, a Service (msm:Service) has an endpoint represented by a URL (rdf:Resource) that exposes one or more operations (msm:Operation) with Input/Output parameters (msm:MessageContents and msmMessagePart); these parameters refer to concepts in an application domain (rdf:Resource). In Figure 1, rounded

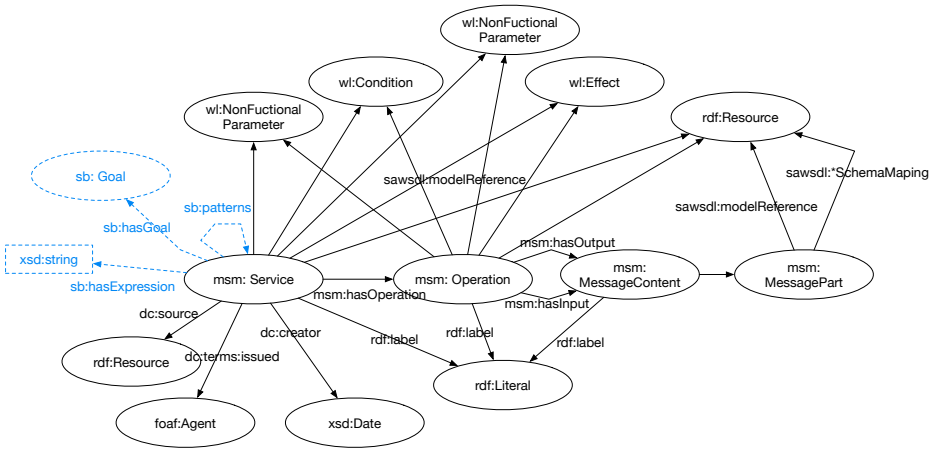


Fig. 3. n MSM ontology extension considering control-flow patterns and guard expressions in order to model service behavior.

rectangles represent concepts (e.g. MSM:SERVICE), arcs represent relationships between concepts (e.g. MSM:HASINPUT), and the squared rectangles represent literals (e.g. XSB:STRING).

In Figure 3, we can see our extension to the MSM service ontology in dotted line and blue colour. We try to be minimalistic in our extension so that it can be applied to other Web service ontologies as well. We use the sb (service behavior) namespace prefix to refer to the elements our proposed extension. Service goals (SB:GOAL) represent the activity that is performed when executing a service, at a high level of abstraction (i.e. is not a service effect) described according to a domain specific ontology. The goal is related to the service through a SB:HASGOAL relationship. Service composition may be restricted according to certain constraints or guard expressions (SB:HASEXPRESION), and services are related to other services through relationships that represent the semantics of control-flow patterns (SB:PATTERNS). In this paper we model six control-flow patterns that are sub-properties of SB:PATTERNS (i.e. they specialize the SB:PATTERNS relationship), each of them represent a relationship between two services: SB:SEQUENCE, SB:ALTERNATIVE, SB:SYNCHRONIZE, SB:DISCRIMINATOR, SB:SELECT and SB:ITERATOR, which are detailed in section 3.3. Some constraints or guard expressions that use the SB:HASEXPRESION relationship can be seen in Figure 2, lines 23 to 25. These are XPath expressions that are traduced to specific control-flow patterns, that is, they contribute to generate an SB:PATTERNS relationship, and the guard expression itself is stored as XSD:STRING related to the service.

With these three specialized relationships and one concept, we are capable of introducing complex control-flow patterns support in the MSM semantic service model. That is, services can relate to each other specifying the type of dependency between them as well as refer to constraints and the goal they pursue. Furthermore, if we consider these elements in addition to the service signature it is possible to determine such relationships automatically. In the following subsection we extend the example presented in subsection 3.1 by including the ontology extension presented in this section. We use the resulting service implementation to illustrate the application of a set of rules, which are also detailed. The rules exploit our ontology extensions to derive control-flow patterns automatically.

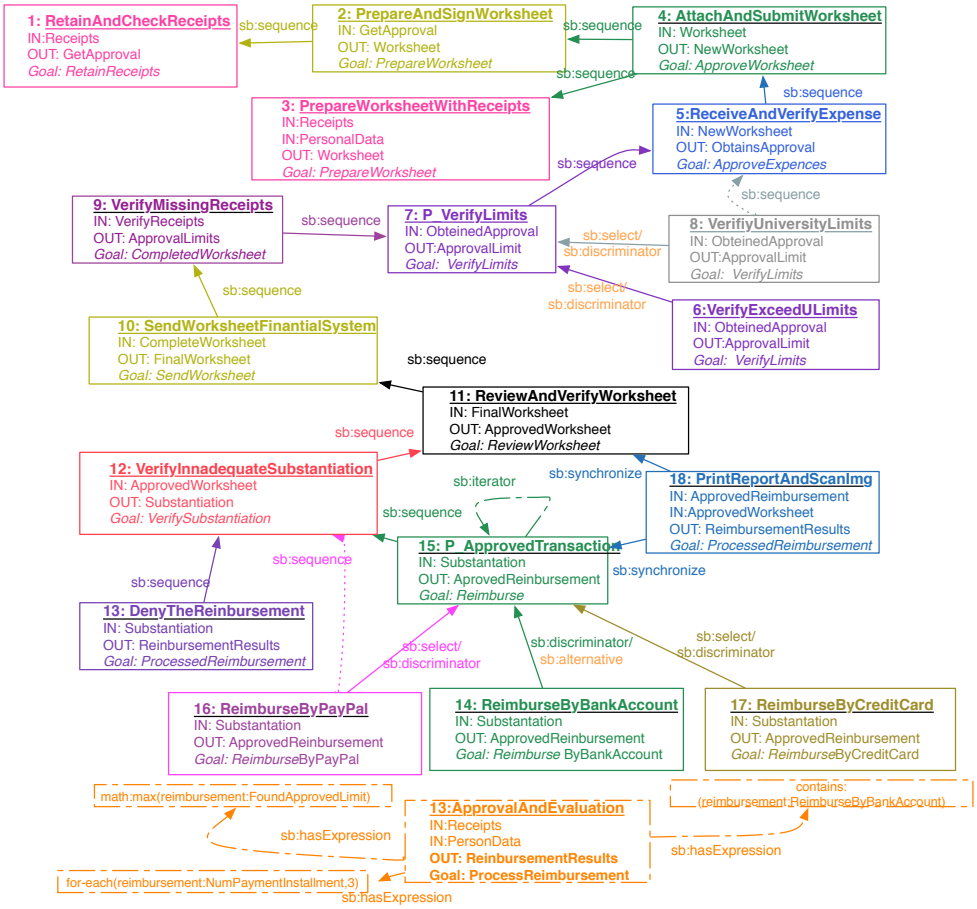


Fig. 4. A composition example for the travel reimbursement scenario: Services are progressively published into our triplestore as indicated by the numbers. The composed service (19) is built from bottom to top (backwards) when a user request is made.

3.3. *Control flow patterns*

In the case of the SAM algorithm [10], the dependencies among atomic services are modelled as an in-memory tree. The SAM algorithm is executed at run-time for each client request. Since the graph of services can grow significantly as companies merge, evolve and change their needs, we pre-compute the possible graph of services dependencies and store the new graph in a specialized database (a NoSQL, graph oriented database). A fragment of the resulting graph will serve as the basis of a new composed service if it is eventually required. Our approach generates new relationships (triples) between services that are stored for later consumption. These relationships are sub-properties of SB:PATTERN represented previous in the ontology (Figure 1).

Figure 4 illustrates a composition graph both at publishing and consuming time for the previously introduced scenario. The control-flow patterns to derive are based on a set of rules, detailed in the remainder of this section.

The widely known *Workflow Patterns Initiative* [50] identifies 43 control-flow patterns divided into 8 categories. In our approach, we consider 3 of 5 basic patterns, 2 out of 14 advanced patterns and 1 pattern of the remaining 6 categories in order to illustrate our approach. They are described below.

3.3.1. *Basic Control Flow Patterns*

In this section patterns capturing elementary aspects of process control are discussed. The patterns we consider are sequence, synchronization, and exclusive choice.

*Pattern 1: Sequence (*SB:SEQUENCE*)*

A sequence pattern models dependencies between services so that a service s2 cannot start before service s1 finishes. An SB:SEQUENCE operator is inferred when the goals of both services (s1 and s2) are different, and service (s1) generates an output, which can be used as an input service (s2).

Figure 4 shows an example including nineteen services, which are progressively published by the provider. The publication process requires service descriptions to be annotated with SAWSDL expressions that are that are taken into account to produce SPARQL 1.1 Update sentences. These sentences generate triples that are stored into a triplestore implemented in Jena. In the example, the process begins right after service 1 (*RetainAndCheckReceipts*) is published. At this point, no relationships are generated since there are no other services. When service 2 (*PrepareandSignWorksheet*) is published a SB:SEQUENCE relationship is generated between both services because they have different goals and *RetainAndCheckReceipts* output matches *PrepareandSignWorksheet*s input.

*Pattern 2: Exclusive Choice (*SB:ALTERNATIVE*)*

Pattern 2 is applied to services with the same goal and output, however; in this case the condition is applied to the goal and optionally to the input parameters. The condition is evaluated to determine which services will be actually invoked and it can be known only when the user issues a request. In our example, an SB:ALTERNATIVE relationship is created between services 14 (*ReimburseByBankAccount*) and 15 (*P_ApprovedTransaction*), since the client requests to pay using a bank account (*Reimburse* goal), which impacts only service 14. The other candidate services, 16 and 17, will be discarded because their goals are different to the request (Figure 3 line 24). In summary, depending on the guard expression, some services may be selected, others may be ignored and new relationships may be created. Expressions applied to services with the same goal and output that evaluate the output results, will cause the inclusion of services related through SB:SELECT relationships. However, expressions that evaluate only the goal (and optionally the input) will cause to ignore those services whose evaluation is negative, such services will be related through an sb:alternative relationships. If no guard expressions are applied, then the services will be related through the sb:discriminator relationship.

*Pattern 3: Synchronization (*SB:SYNCHRONIZE*)*

This pattern is applied when the inputs of a service can be obtained from the outputs of other services, and not a single service can provide all the inputs. In the example, when service

18 (*PrintReportAndScanimg*) is published, this pattern is applied; it requires the execution of service 11 (*ReviewAndVerifyWorksheet*) and service 15 (*P_ApprovedTransaction*) in order to start its execution. Hence, an sb:synchronize relationship is created between service 18 and 11, and service 18 and 15. In the latter case, predecessors (e.g. service 15) are preferred to final services (e.g. services 14, 16 and 17).

3.3.2. *Advanced Branching and Synchronization Patterns*

Advanced patterns refer mainly to parallel invocation. These patterns refer to the various ways that the split and join part of a parallel invocation can arise in business processes. We considered the *Structured Synchronizing Merge*, and the *Structured Discriminator patterns.*

*Pattern 4: Structured Discriminator (*SB:DISCRIMINATOR*)*

In this pattern the thread of control is passed to next service when the first incoming service finishes its execution. That is, only the output of the first service providing a response is considered. When a service is published, the algorithm searches for services with the same goal and output. If there exists more than one service that share the same goal and output, a predecessor node is created (or reused if already exists) and the services are related to the predecessor with an sb:discriminator relationships.

In Figure 4 when the provider publishes service 8 (*VerifyUniversityLimits*) and service 6 (*VerifyExceedULimits*), pattern 4 is applied, since the goal and outputs of both services are the same. Note that service 7 (*P_VerifyLimits*) is the predecessor service created by the system. The same case applies to services 14 (*ReimburseByBankAccount*), 16 (*ReimburseByPayPal*) y 17 (*ReimburseByCreditCard*), which cause the generation of service predecessor 15 (*P_ApprovedTransaction*). Notice that in this case, the goals of services 14 (*ReimburseByBankAccount*), 16 (*ReimburseByPayPal*), and 17 (*ReimburseByCreditCard*) are specializations (inheritance) of the goal of service 15 (*Reimburse*) as defined in the reimbursement ontology. Predecessors are not executable services (empty services); they are generated automatically using the grouped services goal (or the super goal in the case of inheritance) as the predecessor name.

*Pattern 5: Structured Synchronizing Merge (*SB:SELECT*)*

Similarly to pattern 4, services with the same goal and output are grouped together under a predecessor using an sb:select relationship. However, unlike pattern 4, a condition applied to the services output must be evaluated at runtime in order to choose the proper response and such condition can be known only when the user provides a service request. That is, this pattern is not pre-computed at publishing time, but calculated when the consumer issues its request (see Section 5, *Connect2* algorithm).

In Figure 3, when the consumer issues a request and specifies a guard expression on the *FundApprovalLimit* output parameter (line 23), pattern 5 is applied to services 6 (*VerifyExceedULimits*) and 8 (*VerifyUniversityLimits*) since the goal of both services and their output parameters are the same and the output parameter is *FundApprovalLimit*. Since a predecessor was generated in the previous example, the system connects services 6 and 8 with the predecessor with a SB:SELECT relationship.

3.3.3. *Iteration Patterns*

The following pattern deals with capturing repetitive behaviour in a workflow. We only considered the *Structured loop pattern.*

*Pattern 6: Structured Loop (Pre-Test) (*SB:ITERATOR*)*

The iteration pattern occurs when the user requests that a simple or composed service is executed more than once. This need can be only determined from the user request, at composing time, based on the for-each guard expression applied on some goal. In Figure 4, the client specifies that the payment shall be performed in 3 installments only (Figure 3 line 25). Then, an sb:iterator relationship is applied to service 15 (*P_ApprovedTransaction*).

4. COMPO-SWS

In order to test our approach we designed and built *Compo-SWS*, a Web service composer that follows a two sides approach. First, it acknowledges the different roles of the service *publisher* and the service *consumer*, and for the former case it takes advantage of the service availability by pre-calculating all possible relationships, so that, at consuming time, the chances of finding and already identified composed service are higher.

In Figure 5, we summarize the major architectural components of *Compo-SWS*. The dotted rectangle (A) represents the Web services container on the side of the service *publisher*. The publishing process (step 1) requires that the provider interact with *Compo-SWS* interface in order to submit the service description. Such description must be annotated with SA-WSDL expressions in order to be transformed, according to our ontology, into triples (step 2).

The SAWSDL descriptions contain annotations related to the service goal and data types. The *service goal* annotation is an attribute of the WSDLs *portType* element; data types (used as input and output) annotations refer to concepts defined in an external application domain through a *modelReference* element. Figure 6(A) shows a SAWSDL description for the *PrepareWorksheetWithReceipts* service. The services goal is to allow an employee to request a process REIMBURSEMENT (*#PrepareWorksheet*), the service's input includes the receipts (*#Receipts*) and the employee personal data (*#PersonalData*); and the services output is the worksheet (*#Worksheet*) registered by the service. The SAWSDL description is transformed (step 2) into a SPARQL 1.1 Update expression that populates the triplestore. Figure 6(B) presents the SPARQL query generated to populate the triplestore for the SAWSDL description shown in Figure 6(A).

The generated triples are stored into a Jenas TDB triplestore. We use Apache Jena, which is a Java framework providing functionality such as RDF and N3 parsers, and a SPARQL engine among other features. It also provides a programming environment for RDF, RDF(S), OWL, and SPARQL and includes an inference engine based on rules and triplestores. Once translated, the service description is analysed by the *Control Flow Analyser* component, which is responsible of executing the *Connect* algorithm, which connects the services together (step 3 in Figure 12) using the different relationships corresponding to the control-flow patterns. These relationships become new triples that are stored in the database.

The client request can be provided as an XML document (see Figure 2) describing the expected goal and output, and providing some inputs and guards (Figure 5, step 4). The *FindService* algorithm is performed by the *Service Discovery* component (step 5), which

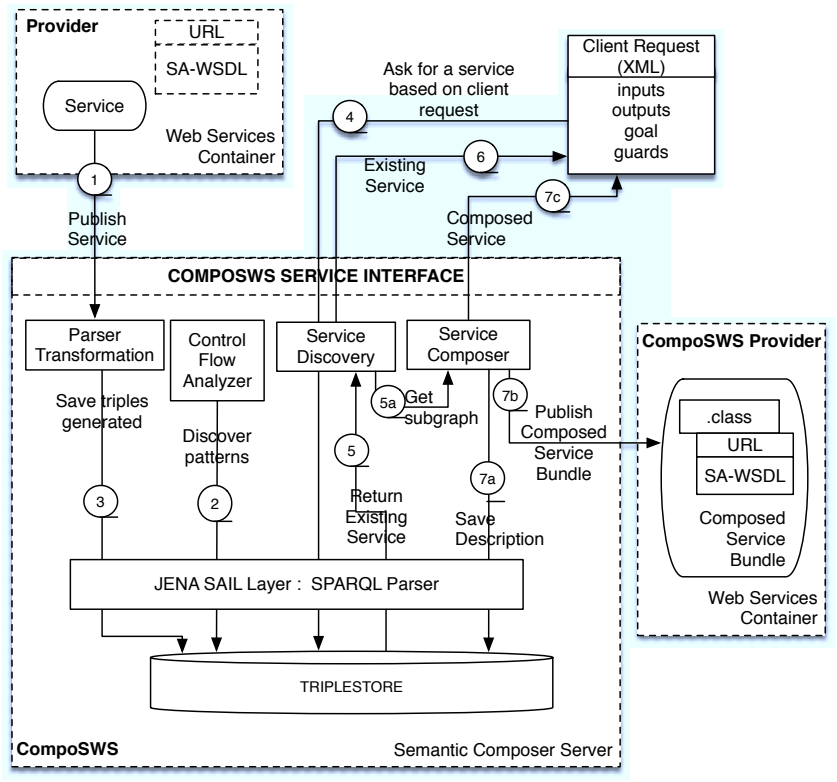


Fig. 5. *Compo-SWS* Architecture.

executes the SPARQL queries on the triplestore. If a service is found, the user is informed (step 6). If no service is found, the *FindService* algorithm is executed (step 5a) by the *Service Composer* component, obtaining a subgraph of services.

The identified subgraph is returned to the user (step 7a) who is asked for approval. If the composed service is approved, a SAWSDL description is created and stored in the triplestore (step 7b). An executable file (Java .class) implementing the component services invocation under the control-flow patterns is created. The bundle, including the executable file and the service description, is deployed on the *CompoSWS Provider Web services* container (step 7c) in order to expose the created services endpoint. The composed services URL is also supplied to the user (step 7d). Our algorithms have been fully implemented in Java and SPARQL using the Jenas SAWSDL4J API and the OWL API as well as the Pellet reasoner as inference engine for logic-based matchmaking. In the following section, the composition algorithms for both sides, the publisher and the consumer, are presented in detail.

5. Composition Algorithms

In this section we present the algorithms that implement the described control-flow patterns, from the publisher and consumer perspective. When a service is published in our platform (Figure 5, steps 1 and 2), the system pre-calculates all the possible relationships between the

Fig. 6. A: SAWSDL description for the *PrepareWorksheetWithReceipts* service; goal, input and output are annotated. B: The N3 query using SPARQL 1.1 Update generated form the SAWSDL.

services (Figure 5, step 3) through the *Connect* algorithm (See subsection 5.1). The resulting graph includes the services goals, input and output characteristics, at the semantic level, including the presented control-flow patterns and rules.

When a consumer requests a service (Figure 5, step 4), the system looks for an existing service (Figure 5, step 5) executing the *Connect2* algorithm. If no service can be found, the system finds a graph fragment that satisfies all or most of the users requests (Figure 5, step 5a). The latter task is accomplished by executing the *FindService* algorithm (See subsection 5.2). If the consumer approves the proposed service, the graph is used as the behaviour (control-flow) of a composed service, which is created, deployed, and publisher later in our system.

5.1. *Pre-computing the graph: the Connect algorithm*

SPARQL is an RDF query language that operates over the data graph model underlying a triplestore. It has some limitations for expressing queries where the length of the path of the consulted graph model is variable, that is, every arc of a graph must be statically and explicitly defined in a SPARQL query. Since we are modelling service dependencies as graphs, our workflows have unpredictable lengths. The *Connect* algorithm addresses this issue by breaking down the graph query in two steps. Figure 7 presents the algorithm following Gooneratne [19].

In lines 2 to 12 (Figure 7), the algorithm uses a SPARQL query to look for the occurrence of the select, discriminator and alternative patterns. That is, it looks for services with a goal and output that is equal to the published services goal and output. For instance, when

```
1  Connect (S) {
2     // FIND DISCRIMINATOR PATTERNS
3     // Search services witch the same goal and output (Q1)
4    C ← FindNode(Goal(S), Output(S))
5    for each S' ∈ C do
6        // find the predecessor of S'
7        P ← Predecessor(S') (Q2)
8        if P is Null then
9              P ← CreatePredecesor(S') (Q3)
10         // connect the service S with P
11       ConnectDiscriminator(S, P);
12   end
13   // FIND SEQUENCE AND SYNCHRONIZE PATTERNS
14   // Search services with different goal and same output of S
15   C ← FindNode(Goal(S), Output(S)) (Q4)
16   for each  S'∈ C do
17     if relationship(S', S) = sequence then
18           connectSequence(S, S');
19     else
20       if relationship(S', S) = synchronization then
21           connectSynchronize(S, S');
22    end
23   I ← FindInput(S)
24   for each I'∈ I do
25     C ← FindNode(Goal(S), I) (Q5)
26     for each S' ∈ C do
27        if relationship(S', S) =  sequence then
28            connectSequence(S, S');
29        else
30          if relationship(S', S) = synchronization then
31             connectSynchronize(S, S');
32    end
   end
```

Fig. 7. Connect algorithm, step 1 (lines 1 to 12) and Connect algorithm, step2 (lines 13 to 30).

service 3 is published, the algorithm searches for services with a *#PrepareWorksheet* goal and *#Worksheet* output (see Figure 8 (Q1)).

The resulting graph is evaluated to determine if all the nodes that share the same goal and output are associated to a predecessor through a SB:SELECT, SB:DISCRIMINATOR, or SB:ALTERNATIVE relationships. Figure 8 (Q2) presents a SPARQL query looking for a predecessor for a specific service (*PrepareWorksheetWithReceipts*). If the predecessor exists but the relationships are missing, nodes and predecessor are connected. If there is not an available predecessor, it is created and the relationships are established (Figure 8 (Q3)).

In lines 13 to 32 (Figure 7), the sequence and synchronize patterns are discovered using two queries (see Figure 8 (Q4 and Q5)). Q4 looks for services with a goal different than the published services goal and with an input parameter that matches the published services output parameter (Figure 8 (Q4)). Q5 looks for services with a goal different than the published services goal and with an output parameter that matches at least one of the published services input parameters (Figure 8 (Q5)).

5.2. *Consuming services*

When a consumer requests a service, the system attempts to find an atomic service providing

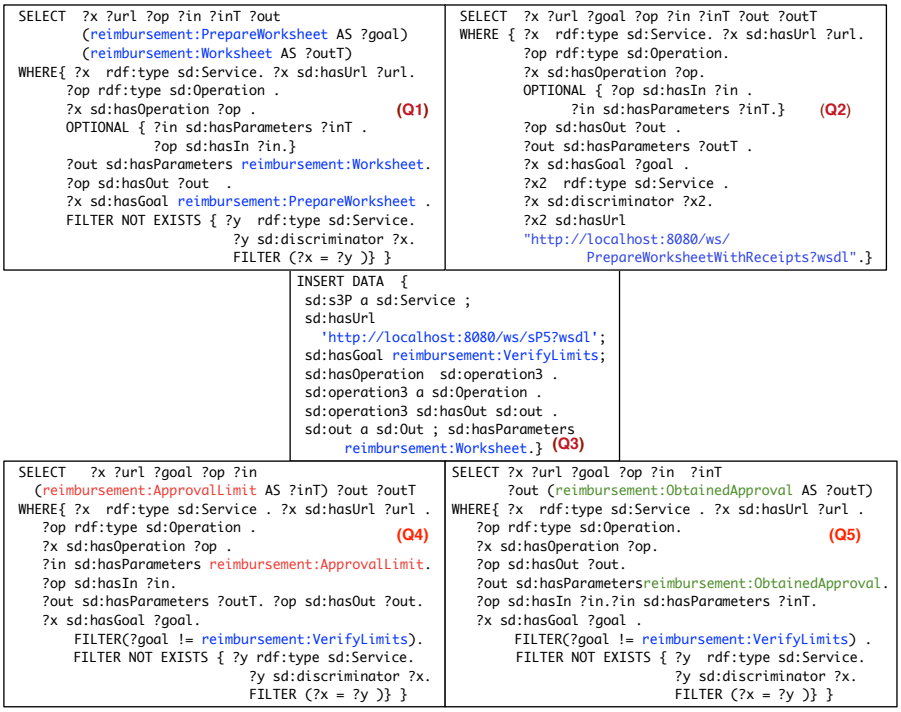


Fig. 8. Q1 query finds all the services with the same goal and output. Q2 query looks for a specific service predecessor. Q3 creates a predecessor in no one is available. Q4 and Q5 look for services with a goal other than the published services goal, in particular (Q4): Finds services with an input parameter that matches the published service output and (Q5): Looks for services with an output parameter that matches one of the published service input parameter.

the requested functionality. Otherwise, the system looks for a subgraph of services of variable length that satisfies clients needs. The subgraph is a set of interrelated services containing all or most of the information provided by the user (input), called *origin* nodes; and containing the expected goal and result (output) required by the user, called *target* nodes. Notice that it may be necessary various services in order to cover all the user requests input parameters, and there may be some parameters that no service in the system support. Our approach minimizes the number of services required to cover the user request, and additional parameters shall be required to the user in an interactive fashion if needed, but such feature is out of the scope of this paper.

For instance, let's consider the example shown in the business scenario previously proposed (See Figure 1). The requested goal is to determine the *Reimbursement Conditions* (*ProcessedReimbursement* goal, *ReimbursementResults* output) given certain *receipts an employee personal data* (*Receipts*, and *PersonalData* input parameters respectively). The user also prefers that the fund maximum approval limit is granted (*FundAppovalLimit*, see line 23 in Figure 2), and the payment option is through a bank account (*ReimburseByBankAccount*, see line 24 in Figure 2), and indicates that 3 (*NumPaymentInstalment*, see line 25 in Figure 2) will be the maximum number of payment instalments. Lets consider as well that only services 1 to 19, as described in the example (see Figure 4), have been published in our

```
1  Connect2 (S) {
2     // FIND ALTERNATIVE AND SELECT PATTERNS
3     // Search services witch the same goal and output (Q1)
4    C ← FindNode(Goal(S), Output(S))
5    for each S' ∈ C do
6        // find the predecessor of S'
7        P ← Predecessor(S') (Q2)
8        if P is Null then
9              P ← CreatePredecesor(S') (Q3)
10         // connect the service S with P
11       ConnectSelectAlternative(S, P);
12   end
13   // FIND ITERATOR PATTERN
14   ConnectIterator(S);
15 end
```

Fig. 9. The *Connect2* algorithm generates sb:select, sb:alternative and sb:iterator relationships as defined by the corresponding control-flow patterns.

system. That is, services 1 to 19 have been related through the SB:SEQUENCE, SB:ITERATOR, SB:SYNCHRONIZE, SB:ALTERNATIVE, SB:SELECT and SB:DISCRIMINATOR control-flow patterns.

Inputs, outputs and goals are described through concepts in an ontology. Some researches [11] exploit the ontology structure and the concept syntax in order to determine a more relaxed similarity degree among concepts, which increases the candidates set. In this paper we consider only exact similarity among concepts since our focus is the composition that takes place once candidates have been found. We plan to include such hybrid approaches as future work. As discussed in Section 3.3, patterns 3 (SB:SELECT), 4 (SB:ALTERNATIVE), and 5 (SB:ITERATOR) can be only applied when the user issues his or her request. That is, the *Connect2* algorithm looks for services with the same goal and output and creates the predecessors if necessary connecting the services with SB:SELECT, SB:ALTERNATIVE or SB:ITERATOR patterns. The *Connect2* algorithm is shown in Figure 9.

Once the *Connect2* algorithm completes the graph, the *FindService* algorithm seeks for an atomic service that matches the requests input, output and goal (line 5, query QF1 in Figure 10). If such service cannot be found, the algorithm searches for the set of nodes that contains the goal and the output defined in the user request that is the set of target nodes, using a SPARQL 1.0 Query (line 8, query QF2 in Figure 10).

Figure 10, lines 11 to 22 is a backtracking algorithm that, starting from a target node (first element of a queue Q), builds a graph until the set of origin nodes are reached. The algorithm incrementally finds services leaving out those that cannot allow it to arrive to a valid solution (i.e. includes only services containing at least one input that matches the user request input I). The resulting graph includes the services related through the defined control-flow relationships.

Considering the patterns of our study, there are only two ways that services can create compositions that include more than one service, that is, either they form a sequence (at least 2 services) or they are invoked in parallel (at least 2 services). These cases correspond to the SB:SEQUENCE and SB:SYNCHRONIZE patterns. The other patterns represent services that are connected either to themselves (SB:ITERATOR) or to an abstract service (predecessor) but have connections among them (SB:DISCRIMINATOR, SB:SELECT, SB:ALTERNATIVE). Since the nodes in the resulting subgraph are interlinked with control-flow relationships, it is

```
1  FindService (S)
2    G ← Goal(S)
3    O ← Output(S)
4    I ← Inputs(S)
5    // Search atomic service with similar goal, inputs and output (QF1)
6    S* ← FindNode(G, I, O)
7    if S* not is Null then
8      C <- FindNode(G, O) (QF2)
9     for each S' ∈ C do
10       Q ← CreateAndEnqueue(S)
11     P = ∅
12      while not is empty Q do
13          S ← FirstInQueue(Q)
14          if S not visited then
15             Visited(S)
16             C ← FindRequireServices(S)
17            for each S' ∈ C do
18                R ← Next(C)
19                if R not visited then
20                     Enqueue(R)
21                     If R has equal input   I  then
22                        P ← P + R
23        S* ← S*  +  P
24     return S*
   end
```

Fig. 10. The *FindService* algorithm is responsible for finding a simple service or discovers the subgraph between a target and origin nodes, generating a subgraph that represents the composed service behaviour.

possible to create a composed service that implements the corresponding logic. In our case, we generate a Java Web Service class that implements the new composed service. That is, the composite is a bundle containing a SAWSDL description and the functional modules (i.e. Java classes) implementing the invocation of services according to the workflow represented by the subgraph. The description contains the set of inputs and the output defined by the user request; it is also stored in our triplestore. It will be possible to generate a BPEL description supporting the proposed control-flow patterns, however such alternative will be considered for future work.

In our example, the origin service is service 3 (*PrepareWorksheetWithReceipts*) because it contains two input parameters (*Receipts* and *PersonalData*) that match the user request input. The subgraph contains related services that include the output parameter (*ReimbursementResult*) and the goal (*ProcessedReimbursement*) as requested by the user. In our example, the algorithm finds one possible solution starting from service 3 (*PrepareWorksheetWithReceipts*) to service 18 (*PrintReportAndScaimg*) passing through services 3, 4, 5, 6, 9, 10, 11, 12, 14 and 18. Hence, a new composite service *ApprovalAndEvaluation* will be created, and the guard expressions (line 23 for services 6 and 8; 24 for service 14; and 25 for service 15 in Figure 2) will be triggered and evaluated at run-time, depending on the user preferences (at run-time), additional control-flow relationships could be created for the composite graph.

6. Evaluation

In this section, we evaluate our approach theoretically, through an analysis of complexity, and experimentally by measuring performance and scalability. Our analysis considers one

```
SELECT  ?url                                                    (QF1)
 WHERE{?x sd:hasOperation ?op.  ?x sd:hasUrl ?url.
     ?in0 sd:hasParameters reimbursement:Receipts.
     ?op sd:hasIn ?in0.
     ?in1 sd:hasParameters reimbursement:PersonalData.
     ?op sd:hasIn ?in1. ?out sd:hasParameters
        reimbursement:ReimbursementResult.
     ?op sd:hasOut ?out.
   ?x sd:hasGoal reimbursement:ProcessedReimbursement. }
```

```
SELECT  ?x ?url ?operation ?in ?inT ?out                        (QF2)
WHERE { ?x sd:hasOperation ?op .
  OPTIONAL { ?in sd:hasParameters ?inT .
             ?op sd:hasIn ?in  .}
  ?out sd:hasParameters. ?x sd:hasUrl ?url.
  ?operation sd:hasOut ?out reimbursement:ReimbursementResult.
  ?x sd:hasGoal reimbursement:ProcessedReimbursement. }
```

Fig. 11. Query (QF1) seeks for an atomic service that matches the requests input, output and goal. Query (QF2) searches for the set of nodes that contains the goal and the output defined in the user request that is the set of target nodes.

operation per service, although it can be extended to include more operations. We also consider a single output parameter and zero or more input parameters.

6.1. *Provider complexity: publishing a new service*

Complexity is calculated considering V, the number of nodes in a graph (services); E, the edges between the nodes (relationships); and k, the number of input parameters for each node. As described before, when a new service is registered in the platform, the possible relationships between services are calculated. The worst-case time complexity analysis of the *Connect()* algorithm, $connect(V, E, k)$, considers three phases, a) finding the nodes matching the new service goals and outputs (line 4 to 13, Figure 5) and b) finding the services with a goal that differs from the new service goal, but has at least one input that matches the output of the new service (line 17 to 25, Figure 7), and c) finding services with a goal different than the new service but with an output that matches the new services inputs (line 27 to 36, Figure 7).

Lets consider M, the number of nodes representing services with the same goal as the new service, and M the number of services with different goal, let be V the total set of nodes, such that $V = M + M$.

For the case of a) the worst-case time complexity analysis occurs when $M = V$, that is all the nodes matches the new service goal, hence, the order of this step is calculated as $TConnect(V, E, k) = V$, that is the process of creating a relationship between the new service and the previously existing services. For the case of b), the worst-case scenario occurs when the new service output matches all the previously stored services input, in this case, the order is $TConnect(V, E, k) = V$. For the case of c), the worst-case scenario occurs when given the new services k inputs, every V node output matches the new services input, hence the order is $TConnect(V, E, k) = V * k$. That is, for each input of the new service, a relationship is established with all the existing nodes. Therefore in the worst case, the algorithm has order

$TConnect(V, E, k) = V * k$ time complexity. Hence, the algorithm is lineal.

6.2. *Consumer complexity: atomic or composed (on the fly) service*

When consuming a service, the algorithm *FindService()* recursively finds a graph of services providing the desired functionality. The worst-case time complexity for *FindService()* is defined as $TFindService(V, E, k) = 1$, that is, it performs a query searching for an atomic service that matches users criteria (Figure 9, QF1, line 3). If there is no atomic service, the algorithm will perform also a single query searching for the services matching the users request goal and output. In this case the time complexity is calculated as $TFindService(V, E, k)$ the query result will include a list of nodes $N < V$, the algorithm performs a depth-first recursive search. The end of the recursion occurs when a nodes or a set of nodes inputs (k) matches the user requirements inputs. The worst case time complexity of the depth-first search is E (all the edges) and, since this search must be performed for all the results obtained in the previous query the time complexity is $TCreatePath(V, E, k) = E * N$. Hence, the order of complexity for the consumer phase is $E * N$ time complexity, that is, $O(N^2)$ complexity.

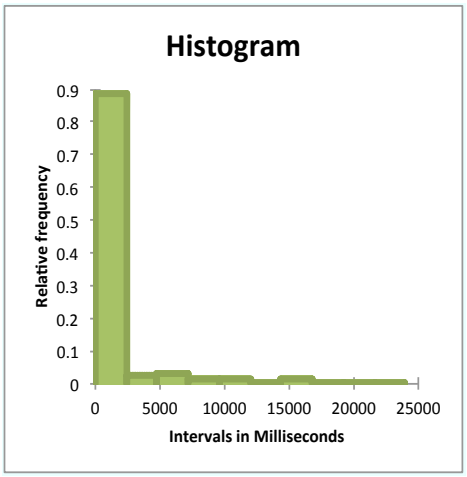
6.3. *Experimental evaluation*

In order to measure the performance and scalability of our approach, we used a SAWSDL test-bed collection semi-automatically derived from a SAWSDL public dataset (SAWSDL- TC3 WSDL11). Descriptions were annotated with a goal concept since the collection considered only input/output concepts. The original collection consisted of 1080 Web services covering different application domains: education, medical care, food, travel, communication, economy and weaponry. We only used 980 services for this test and discarded all services with no outputs. We ran our experiments on an Intel Xeon E5620 with 2,4 Ghz 4Core and 3 GB RAM, running on Linux Ubuntu 11.04. We performed the tests 10 times and we averaged all the results in order to obtain a reliable measure. We evaluated the pre-computing response time when publishing a new service (the connect algorithm) and the response time when requesting a service (service discovery and composition).

6.3.1. *Performance analysis: Publishing time*

We measured the time it takes to add a new service to the graph, varying the number of web services from 1 to 980. In order to avoid additions with no effects (no relationships) we added first the biggest set of unique nodes arranged in a deep relationship (i.e. sequence or synchronize) conforming a composite. In our dataset, the largest possible composed service corresponds to a set of four nodes connected with a sequence relationship (three edges). We added these services first and the remainder nodes were added in random order, one by one. The experiment was run 10 times and the results averaged.

In Figure 12 the response time obtained when publishing services is shown as a histogram of 10 intervals; tables a) and b) presents some descriptive analysis. The response time is 0 for a total of 79% of the added services; this result varies from 69% to 94% according to the applied pattern. The average response time is 7 milliseconds, again varying according to the pattern, the maximum response time (average) obtained is 105 milliseconds corresponding to the addition of a service that causes the generation of various select (or discriminator)



(a)

	Lower bound	Upper bound	Frequency	Relative frequency
1	0.0	2.4	866	0.885
2	2.4	4.8	24	0.025
3	4.8	7.2	32	0.033
4	7.2	9.6	12	0.012
5	9.6	12.0	15	0.015
6	12.0	14.4	4	0.004
7	14.4	16.8	17	0.017
8	16.8	19.2	5	0.005
9	19.2	21.6	2	0.002
10	21.6	24.0	2	0.002

(b)

			Response time		
Pattern	0's	>0	Average (ms)	St. Dev. (ms)	Max (ms)
Sequence	86%	14%	12	32	128
Select	69%	31%	8	14	110
Discriminator	69%	31%	8	14	110
Synchronize	94%	6%	2	8	73
Total	79.5%	20.5%	7	17	105

Fig. 12. Descriptive analysis of the performance results when publishing services.

relationships. The standard deviation is about 17 milliseconds, which is, three times the average.

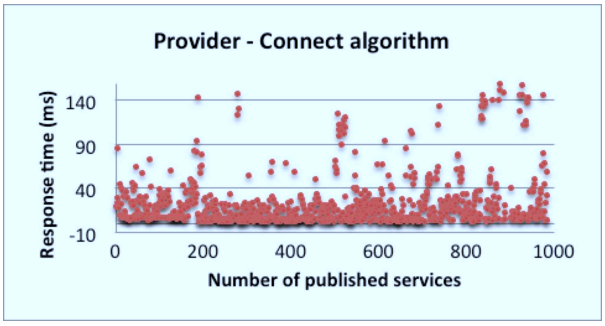


Fig. 13. Accumulated maximum response time obtained from the connect algorithm.

Figure 14 shows the time response results (y-axis) when publishing the Web services (x-axis), in seconds. The figure shows the distribution of such values. As can be seen in the figure, the time for pre-computing services composition increases with the number of web services, this is explained since the more available services, the more comparisons must be performed and probably the more relationships must be created. Notice also that the select and discriminator patterns have the same behaviour; this is because both relations are created at the same time. In addition, the alternative relationship is created only when a consumer requests this relation, hence it was not included in our analysis.

6.3.2. *Performance analysis: Consumer time*

We measured the response time needed to process Web services requests. Following a similar strategy, we published first a set of four connected services and then added the remainder

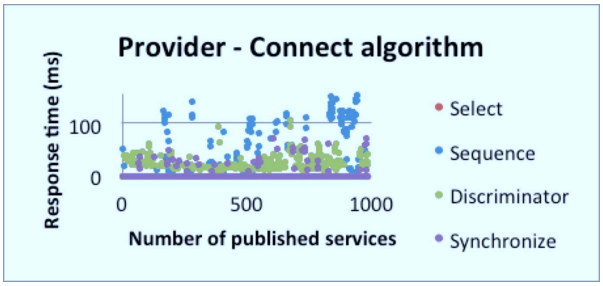


Fig. 14. Response time considering the sequence, select, discriminator, and synchronize patterns.

services randomly. We performed queries asking for services that we knew included 1 (SS), 2 (2CS), 3 (3CS) and 4 (4CS) services, however, we did not stored the composed services into the triplestore (so that, they need to be discovered every time a query is performed). The experiment was ran 10 times and the results averaged. Figure 14 shows the mean execution time required for processing the queries; as we can observe the response time increases as the number of Web services in the triplestore increases. This is explained because we perform deep and breadth searches, so that, the more services are published, the more likely they conform complex composites and hence the time spent by the createPathComposedService algorithm increases.

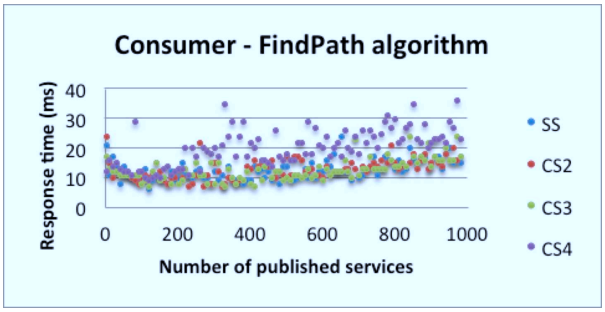


Fig. 15. Response time when searching for atomic and composed services including 1 (SS), 2 (2CS), 3 (3CS) and 4 (4CS) services.

6.3.3. *Performance evaluation and metrics*

We compared our approach using eight public repositories from Web Service Challenge 2008 [9], CompoIT [43, 44] and WSD [35], since the datasets present the same size and perform a similar task. However, notice that service composition implemented in such approaches correspond to simple control-flow patterns, namely sequence and alternative, requiring deep search instead of both deep and breadth search, which is our case. In addition, the WSC challenges as well as WSD perform only the search task, leaving out the composition step (and time); CompoIT and WSD considers search by similarity whereas we are limited to exact matches which causes that they can obtain a high number of composed services while we are limited to exact matches. Figure 16(a) summarizes the results of these approaches

in terms of the number of relevant services (Serv) number of discovered services used in the generated service compositions, dynamic service composition or discovery time (Time(ms)) in milliseconds; that is, the time required to process a user service request and perform the discovery of services and composition if possible. Figure 16(b) presents a comparison of the top-8 approaches. The number of I/O parameters however is around 5700 (taking into account semantic concepts) for WSC while we keep 7 I/O parameters.

The quality of each composition includes also the complexity of the composed services. The depth of a composed service in the WSC dataset falls between 5 to 8, comprehending also 10 to 20 services, whereas the deepest composed service in our dataset includes 5 composition layers and 43 services. However, our composed services include the sequences/synchronize pattern (depth) as well as the select/discriminator pattern (breadth). For the latter case, the broadest composed service includes 21 services.

Aproach - DataSet	#Serv	Time (ms)	Sol (serv/length)
CompoSWS - SAWSDL-TC3	1,080	37	sol 43/ 5
CompoIT - WSC'04	1,041	101	sol 10 / 5
CompoIT - WSC'08	1,090	180	sol 20 / 8
Groningen - WSC'04	1,041	219	sol 10 / 5
Tsinghua - WSC'05	1,090	250	sol 20 / 8
Tsinghua - WSC'04	1,041	312	sol 10 / 5
Kassel - WSC'04	1,041	828	sol 10 / 5
WDS - Xmethod	873	1,800	
Groningen - WSC'05	1,090	14,734	sol 20 / 8
Pennsylvania - WSC'04	1,041	28,078	sol 10 / 5
Kassel -- WSC'05	1,090	300,219	sol 20 / 8
Pennsylvania - WSC'05	1,090	726,078	sol 20 / 8

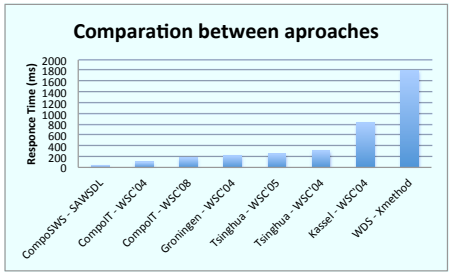


Fig. 16. A comparison, based on number of services composed or discovered versus response time and the quality of the solution, among our approach (CompoSWS) and the WSC challenge, CompoIT and WDS.

7. Conclusions

In this paper we propose a technique for automatically deriving simple and complex composed service behaviour from the component services characteristics, dynamically. In order to deal with the resulting complexity, we also propose a strategy for pre-computing possible relationships resulting in a control-flow graph. Later, queries can identify graph fragments as potential candidates for a complete or partial composed service, automatically and dynamically. Our results provide good evidence of the potential of our approach. Despite the increasing response time at publishing-time, 75% of such responses took almost 0 seconds. Regarding the consumer-time, our observations testify that as the composites are stored, the service response time also decreases. An important limitation of our approach is the need for providers, and consumers to know beforehand the ontologies describing the concepts associated with inputs, outputs and goals as well as properly writing the request and annotating the services. Possible solutions for such challenges include the emergence of popular ontologies in various niches such as FOAF describing social relationships, *Good Relations* describing *e-Commerce*, among others.

In this paper we used semantic Web technologies but we placed an emphasis on the graph nature of the data model rather than the semantic aspects. SPARQL 1.1 presents some limitations to perform queries such as those needed in our work but other NoSQL databases and languages such as Neo4J, Cipher, and Gremlin may serve to provide an alternative imple-

mentation with better performance. In addition, we exploited only concept specialization in order to implement goal queries as described in the ontologies, however, other techniques that range from logical (plugin, subsumed-match, subsumed-by-match) to statistical (similarity by nearest neighbour, pearson, jacquard, etc.) or a hybrid, will be applied as future work. In such cases, we expect an explosive growth in the number of relationships between services and possible a degrading performance and scalability. Finally, we just explored 6 control-flow patterns out of the numerous existing and ones in order to prove the feasibility of our approach, this work should be extended to determine the feasibility of automatically deriving the remaining patterns.

References

1. Agarwal, S. (2007) , *Formal description of web services for expressive matchmaking*, (Doctoral dissertation, Karlsruhe Institute of Technology).
2. Ahmad, H., & Dowaji, S. (2013) , *Linked-owl: A new approach for dynamic linked data service workflow composition*, Webology, 10(1).
3. Alarcon, R., & Wilde, E. (2010, April) , *Linking data from restful services*, In Third Workshop on Linked Data on the Web, Raleigh, North Carolina (April 2010).
4. Alowisheq, A., Millard, D. E., & Tiropanis, T. (2009), *Express: Expressing restful semantic services using domain ontologies*, In The semantic web-iswc 2009 (pp. 941–948). Springer.
5. Arenas, M., Conca, S., & Prez, J. (2012), *Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard*, In Proceedings of the 21st international conference on world wide web (pp. 629638).
6. Aversano, L., & Taneja, K. (2006), *A genetic pro- gramming approach to support the design of service compositions*, International Journal of Computer Systems Science & Engineering, 21(4), 247–254.
7. Bener, A. B., Ozadali, V., & Ilhan, E. S. (2009) , *Semantic matchmaker with precondition and effect matching using swrl*, Expert Systems with Applications, 36(5), 9371–9377.
8. Bellido, J., Alarcon, R., & Pautasso, C. (2013) , *Control-flow patterns for decentralized restful service composition*, ACM Trans. Web, 8(1), 5:1–5:30. Retrieved from http://doi.acm.org/10.1145/2535911 doi: 10.1145/ 2535911
9. Bansal, A., Blake, M. B., Kona, S., Bleul, S., Weise, T., & Jaeger, M. C. (2008, July). , *WSC-08: continuing the web services challenge*, In E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on (pp. 351-354). IEEE.
10. Brogi, A., Corfini, S., & Popescu, R. (2008), *Semantics-based composition-oriented discovery of web services*, ACM Transactions on Internet Technology (TOIT), 8(4), 19.
11. Chabeb, Y., Tata, S., & Ozanne, A. (2010) , *Yasa-m: A semantic web service matchmaker*, In Advanced information networking and applications (aina), 2010 24th ieee international conference on (pp. 966–973).
12. Chinnici, R., Moreau, J., Ryman, A., & Weerawarana, S. (2009) , *Web services description language (wsdl)*, version 2.0, part 1: Core language, w3c recommendation, 16 june 2007. World Wide Web Consortium (W3C), URL: http://www. w3. org/TR/2007/REC-wsdl20-20070626/(04.02. 2008).
13. De Bruijn, J., Lausen, H., Polleres, A., & Fensel, D. (2006), *The web service modeling language wsml: An overview*, In The semantic web: Research and applications (pp. 590–604). Springer.
14. Dietze, S., Benn, N., Yu, H. Q., Pedrinaci, C., Makni, B., Liu, D., and Domingue, J. (2010) , *Comprehensive service semantics and light-weight Linked Services: towards an integrated approach*
15. DaMello, D. A., Ananthanarayana, V., & Salian, S. (2011), *A review of dynamic web service composition techniques*, In Advanced computing (pp. 85–97). Springer.

16. Domingue, J., Galizia, S., & Cabral, L. (2005) , *Choreography in irs-iiicoping with heterogeneous interaction patterns in web services*, In The semantic webiswc 2005 (pp. 171–185). Springer.
17. Dustdar, S., & Schreiner, W. (2005), *A survey on web services composition*, International Journal On Web and Grid Services, 1(1), 1–30.
18. Ghafarian, T., & Kahani, M. (2009, July), *Semantic web service composition based on ant colony optimization method*, In Proceedings of the First International Conference on Networked Digital Technologies (pp. 171-176). Washington, DC: IEEE Computer Society.
19. Gooneratne, N., Tari, Z., & Harland, J. (2007, March) , *Verification of web service descriptions using graph-based traversal algorithms*, In Proceedings of the 2007 ACM symposium on Applied computing (pp. 1385-1392). ACM.
20. Hadley, M. (2009, August) , *Web application description language*, World Wide Web Consortium, Member Submission SUBM-wadl-20090831.
21. Hamadi, R., & Benatallah, B. (2003) , *A petri net-based model for web service composition*, In Proceedings of the 14th australasian database conference-volume 17 (pp. 191–200).
22. He, J., Zhang, Y., Huang, G., & Cao, J. (2012, February) , *A smart web service based on the context of things*, ACM Trans. Internet Technol., 11(3), 13:1–13:23. Retrieved from http://doi.acm.org/10.1145/2078316.2078321 doi: 10.1145/2078316.2078321
23. Hoffmann, J., Bertoli, P., & Pistore, M. (2007), *Web service composition as planning, revisited: In between background theories and initial state uncertainty*, In Proceedings of the 22nd National Conference of the American Association for Artificial Intelligence (pp. 1013-1018). Palo Alto, CA: AAAI.
24. Kil, H., & Nam, W. (2013) , *Semantic web service composition via model checking techniques*, International Journal of Web and Grid Services, 9(4), 339–350.
25. Klein, M., Konig-Ries, B., & Mussig, M. (2005) , *What is needed for semantic service descriptions? a proposal for suitable language constructs*, International Journal of Web and Grid Services, 1(3), 328–364.
26. Klusch, M., Gerber, A., & Schmidt, M. (2005), *Semantic Web service composition planning with OWLS-Xplan*, In Proceedings of the AAAI Fall Symposium on Semantic Web and Agents, Arlington, VA. Palo Alto, CA: AAAI.
27. Klusch, M., Kapahnke, P., & Zinnikus, I. (2009) , *Hybrid adaptive web service selection with sawsdl-mx and wsdl-analyzer*, In The semantic web: Research and applications (pp. 550–564). Springer.
28. Kopecky, J., Vitvar, T., Bournez, C., & Farrell, J. (2007), *Sawsdl: Semantic annotations for wsdl and xml schema.*, Internet Computing, IEEE, 11(6), 60–67.
29. Kylau, U., Stollberg, M., Weber, I., & Barros, A. (2012), *Service functionality and behavior. In Handbook of service description*, (pp. 269–293). Springer.
30. Krummenacher, R., Norton, B., & Marte, A. (2010), *Towards linked open services and processes*, In Future Internet-FIS 2010 (pp. 68-77). Springer Berlin Heidelberg.
31. J. Lathem, K. Gomadam, and A. P. Sheth , *Sa-rest and (s)mashups : Adding semantics to restful services*, In Proceedings of the First IEEE International Conference on Semantic Computing, pages 469–476, 2007.
32. Maleshkova, Maria; Pedrinaci, Carlos and Domingue, John (2009) , *Supporting the creation of semantic RESTful service descriptions*, In: 8th International Semantic Web Conference (ISWC 2009), 25-29 Oct 2009, Washington D.C., USA.
33. Maleshkova, M., Kopecka, J., & Pedrinaci, C. (2009, January) , *Adapting SAWSDL for semantic annotations of restful services*, In On the Move to Meaningful Internet Systems: OTM 2009 Workshops (pp. 917-926). Springer Berlin Heidelberg.
34. McGuinness, D. L., Van Harmelen, F., et al. (2004) , *Owl web ontology language*, W3C recommendation, 10.
35. Nayak, R., & Bose, A. (2015) , *A Data Mining Based Method for Discovery of Web Services and their Compositions*, In Real World Data Mining Applications(pp. 325-342). Springer International Publishing.

36. Pautasso, C. (2009a) , *Composing restful services with jopera*, In A. Bergel & J. Fabry (Eds.), Software composition (Vol. 5634, p. 142-159). Springer Berlin Heidelberg.
37. Pautasso, C. (2009c, September) , *Restful web service composition with bpel for rest*, Data Knowl. Eng., 68(9), 851-866. Retrieved from http://dl.acm.org/ citation.cfm?id=1550965.1551240 doi: 10.1016/j.datak.2009.02.016
38. Pautasso, C. (2009) , *RESTful Web service composition with BPEL for REST*, Data & Knowledge Engineering, 68(9), 851-866.
39. Pedrinaci, C., Lambert, D., Maleshkova, M., Liu, D., Domingue, J., & Krummenacher, R. (2011), *Adaptive service binding with lightweight semantic web services*, In Service engineering (pp. 233260). Springer
40. Pedrinaci, Carlos, et al. , *iServe: a linked services publishing platform*, CEUR workshop proceedings. Vol. 596. 2010.
41. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., & Traverso, P. (2004), *Planning and monitoring web service composition*, In C. Bussler & D. Fensel (Eds.), Proceedings of the 11th International Conference on Artificial Intelligence: Methodology, Systems and Applications (LNCS 3192, pp. 106-115).
42. Rodrguez-Mier, P., Mucientes, M., Lama, M., & Couto, M. I. (2010), *Composition of web services through genetic programming*, Evolutionary Intel- ligence, 3(3-4), 171–186. doi:10.1007/s12065-010- 0042-z
43. Rodrguez-Mier, P., Mucientes, M., Vidal, J. C., & Lama, M. (2012), *An optimal and complete algorithm for automatic web service composition*, International Journal of Web Services Research (IJWSR), 9(2), 1-20.
44. Rodriguez Mier, P., Pedrinaci, C., Lama, M., & Mucientes, M. (2015) *An Integrated Semantic Web Service Discovery and Composition Framework.*
45. Rosenberg, F., Curbera, F., Duftler, M. J., & Khalaf, R. (2008). *Composing restful services and collaborative workflows: A lightweight approach*, Internet Computing, IEEE, 12(5), 24-31.
46. Russell, N., Ter Hofstede, A. H., & Mulyar, N. (2006), *Workflow controlflow patterns: A revised view*
47. Russell, N. C., Aalst, W. M. van der, & Hofstede, A. H. ter. (2009), *Designing a workflow system using coloured petri nets*, In Transactions on petri nets and other models of concurrency iii (pp. 1–24). Springer.
48. Sirin, E., Parsia, B., Wu, D., Hendler, J., & Nau, D. (2004), *HTN planning for Web service composition using SHOP2*, Web Semantics: Science Services and Agents on the World Wide Web, 1(4), 377–396. doi:10.1016/j.websem.2004.06.005
49. Ter Hofstede, Arthur HM, et al. , *Modern Business Process Automation: YAWL and its support environment*, Springer Science & Business Media, 2009.
50. van Der Aalst, W. M., Ter Hofstede, A. H., Kiepuszewski, B., & Barros, A. P. (2003), *Workflow patterns. Distributed and parallel databases*, 14(1), 5-51.
51. Verborgh, R., Steiner, T., Deursen, D., Van de Walle, R., & Valles, J. (2011, oct.). , *Efficient runtime service discovery and consumption with hyperlinked restdesc*, In Next generation web services practices (nwesp), 2011 7th international conference on (p. 373 -379). doi: 10.1109/NWeSP.2011.6088208
52. Verborgh, R., Steiner, T., Van Deursen, D., Coppens, S., Valles, J. G., & Van de Walle, R. (2012) , *Functional descriptions as the bridge between hypermedia apis and the semantic web*, In Proceedings of the third international workshop on restful design (pp. 3340). New York, NY, USA: ACM. Retrieved from http://doi .acm.org/10.1145/2307819.2307828 doi: 10.1145/2307819.2307828
53. Vitvar, T., Kopecky, J., Zaremba, M., & Fensel, D. (2007, November), *Wsmo-lite: Lightweight semantic descriptions for services on the web* , In Web Services, 2007. ECOWS'07. Fifth European Conference on (pp. 77-86). IEEE.
54. Vitvar T., Kopecky J, Viskova J., and Fensel D. , *WSMO-Lite Annotations for Web Services. In the Semantic Web: Research and Applications*, ESWC 2008
55. Xu, J., Chen, K., & Reiff-Marganiec, S. (2011), *Using Markov decision process model with logic*

scoring of preference model to optimize HTN Web services composition, International Journal of Web Services Research, 8(2), 53–73. doi:10.4018/jwsr.2011040103

56. Zhang, Y., Zhang, X., & Liu, F. (2010), *Semantic web service matchmaking based on service behavior. In Anti-counterfeiting security and identification in communication (asid)*, 2010 international conference on (pp. 184–188).