A Multilevel Composability Model for Semantic Web Services

Brahim Medjahed, *Member*, *IEEE*, and Athman Bouguettaya, *Senior Member*, *IEEE*

Abstract—We propose a *composability model* to ascertain that Web services can safely be combined, hence avoiding unexpected failures at runtime. Composability is checked through a set of *rules* organized into four *levels*: *syntactic*, *static semantic*, *dynamic semantic*, and *qualitative* levels. We introduce the concepts of *composability degree* and τ-*composability* to cater for *partial* and *total* composability. We also propose a set of algorithms for checking composability. Finally, we conduct a performance study (analytical and experimental) of the proposed algorithms.

Index Terms—Web service, semantic Web, ontology, service composition, composability.

1 Introduction

The Web has been a powerful tool to elicit connectivity to a wealth of information that had been previously inaccessible. However, as the number of data sources and applications available on the Web increased tremendously, it has become apparent that it could no longer sustain its growth in its present form. A large proportion of today's data on the Web is mostly "understandable" by humans or custom developed applications. The main impediment has been adding *semantics*, hence enabling machines to "understand" and automatically process the data that they merely display at present. The *Semantic Web* is an emerging paradigm shift to fulfill this goal. It is an extension of the existing Web, in which information is given well-defined meaning [3].

The development of concepts and techniques to support the envisioned Semantic Web is the priority of various research communities. *Web service* and *ontology* are two concepts that are taking the spotlight in enabling tomorrow's Web (i.e., Semantic Web) [1], [12], [13]. A *Web service* is as a set of related functionalities that can be programmatically accessed through the Web. Examples of Web services span several application domains including *e-government* (e.g., e-tax preparation) and *B2B E-commerce* (e.g., stock trading). An *ontology* is a *formal* and *explicit* specification of a *shared conceptualization* [3]. Ontologies are expected to play a central role to empower Web services with semantics. The combination of these powerful concepts (i.e., Web services and ontologies) has resulted in the emergence of a new generation of Web services called *Semantic Web services*.

- *B. Medjahed is with the Department of Computer and Information Science, University of Michigan-Dearborn, 4901 Evergreen Road, Dearborn, MI 48128. E-mail: brahim@umich.edu.*
- *A. Bouguettaya is with the Department of Computer Science, Virginia Tech, 7054 Haycock Road, Falls Church, VA 22043. E-mail: athman@vt.edu.*

Manuscript received 16 Feb. 2004; revised 22 Sept. 2004; accepted 26 Jan. 2005; published online 18 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0051-0204.

The landscape created by Semantic Web services has spurred several research issues. One important challenge is *service composition* which refers to the process of combining different Web services to provide a *value-added* service [18]. Service composition is emerging as *the* technology of choice for building cross-enterprise applications on the Web [1], [13]. This is mainly motivated by three factors. First, tomorrow's Web is expected to be highly populated with Web services. Second, the adoption of XML-based messaging over well-established protocols (e.g., HTTP) enables communication among disparate systems. Third, the use of a document-based messaging model in Web services caters for loosely coupled relationships among organizations' applications.

Web service composition involves two types of services: *simple* and *composite*. *Simple* services are Internet-based applications that do not rely on other Web services to fulfill consumers' requests. An example of a simple Web service is a `translator` that accepts words in a given language (e.g., Chinese) and returns their translation in another language (e.g., English). A *composite* service is defined as a conglomeration of outsourced Web services (called *participants*) working in tandem to offer a *value-added* service. `Tax Preparator` is an example of composite service used by citizens to file their taxes. It includes several participants such as financial services at citizens' companies to get W2 information, banks' and investment companies' services to retrieve investment information, and electronic tax filing services provided by state and federal revenue agencies [18].

A large body of research has recently been devoted to Web service composition. Several techniques and prototypes have been proposed by the research community (e.g., [2], [5], [10]). Standardization efforts are also under way for supporting service composition (e.g., *BPEL4WS* [1], [13]). However, these techniques, prototypes, and standards provide little or no support for the semantics of Web services, their messages, and interactions. Additionally, they generally require dealing with low-level details, thus making the service composition error-prone and time-consuming. To illustrate the complexity of the composition process, let us consider the example of users willing to

translate a word from Chinese to Urdu. Assume that no `Chinese-Urdu` translation service is available. One solution would be to combine two simple services $WS_1 =$ `Chinese-English` and $WS_2 =$ `English-Urdu`. The tasks performed by users to compose WS_1 and WS_2 include the following. They first have to determine which services are relevant to their requests (i.e., WS_1 and WS_2). For that purpose, they need to delve into a large space of heterogeneous services. Those services are related to different domains of interest such as insurance, translation, and stock market. Users then should understand the exact format, content, and semantics of messages exchanged between WS_1 and WS_2. They must also "manually" specify the way WS_1's and WS_2's messages are mapped to each other. Finally, they should find out how WS_1 and WS_2 can together define an overall business process (e.g., define the order of messages).

As demonstrated by the previous example, service composition involves going through several complex stages. One important, yet tedious, stage is the *composability* of interacting services. *Composability* refers to the process checking whether participant services can actually work together, hence avoiding unexpected failure at runtime. The "manual" checking of service composability would clearly be unrealistic on the envisioned Semantic Web. What is needed is a framework where composability would be checked *automatically* and transparently. In this paper, we propose a *composability model* for semantic Web services. Composability is checked through a set of *rules* organized into four *levels*: *syntactic*, *static semantic*, *dynamic semantic*, and *qualitative* levels. Each rule compares a specific pair of *attributes* of interacting Web services. We also define the notions of *composability degree* and τ-*composability* to cater for *partial* and *total* composability. We also propose a set of algorithms for checking Web service composability. Finally, we conduct a performance study (analytical and experimental) of the proposed algorithms.

The remainder of this paper is organized as follows: Section 2 introduces an e-government case study. Section 3 describes the proposed composability model. Section 4 presents our approach for the semantic description of Web services. Section 5 gives details about semantic composability rules. Section 6 proposes algorithms for checking composability and presents the analytical model. Section 7 focuses on the implementation and performance analysis of the proposed algorithms. Section 8 gives an overview of the related work. Section 9 provides concluding remarks.

2 Case Study: E-Government Web Services

While our approach is generic enough to be applicable to a wide range of applications, we use the area of *e-government* as a case study. In particular, we focus on social services for senior citizens. These services may be used "individually" or combined together to provide value-added services. Assume that a case worker *Mary* is planning to organize a visit to a *Senior Activity Center* or SAC (club for senior citizens). *Mary*'s request includes several *subrequests* (Fig. 1). *Mary* first retrieves the list of citizens interested in visiting SAC (SR_1). Assume that *Mary* gets the names and zip codes of those citizens instead of their full addresses. *Mary* then sets an appointment to visit the SAC (SR_2). Once a visit is scheduled, she gets driving directions from each citizen's location to the SAC (SR_3). She finally notifies each citizen about the schedule and the driving directions to the SAC (SR_4).

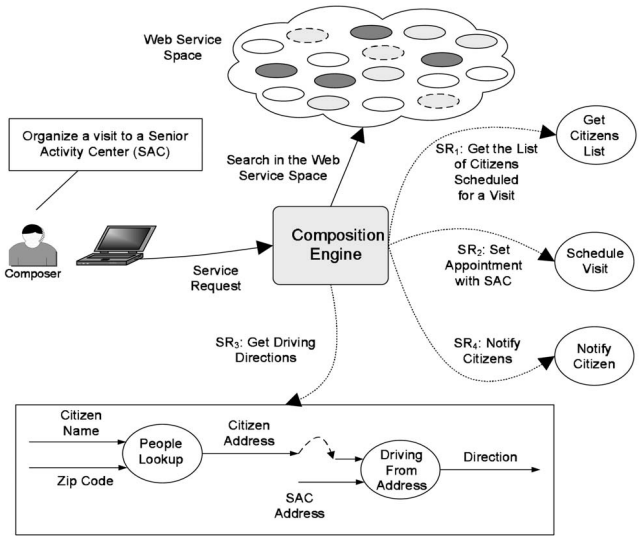


Fig. 1. Composability of Web services.

Each subrequest is typically performed by invoking one or more services. The *composition engine* would delve into the service space to determine participants that "best" serve each subrequest (Fig. 1). Simple services are found relevant to subrequests SR_1, SR_2, and SR_4. Assume now that the "Get Driving Directions" (SR_3) returns the driving directions given a citizen's name, zip code, and address of the SAC. Since there is no simple service that offers such functionality, one solution would be to compose existing services in a way that would transparently fulfill SR_3.

To execute *Mary*'s request, the composition engine first needs to "understand" the *semantics* of existing Web services. It should, for example, "understand" that `Direction-From-Address` provides "directions between two addresses," and, hence, cannot be used alone to perform SR_3. The composition engine finds the `People-Lookup` simple service as relevant to SR_3. Indeed, `People-Lookup` returns citizens' addresses, given their names and zip codes. Hence, combining `People-Lookup` and `Direction-From-Address`, as depicted in Fig. 1, would allow the execution of SR_3. To enable such composition, the engine checks that the way participants are composed together is "correct." It needs, for example, to make sure that the format, content, and semantics of messages exchanged between `People-Lookup` and `Direction-From-Address` are "compatible." In the rest of this paper, we propose a *composability model* to check the "compatibility" of participant services.

3 The Web Service Composability Model

The semantic description of Web services is an important requirement for checking their composability. The large scale and heterogeneity of Web services may hinder any attempt for "understanding" their semantics and, hence,

composing them. We define a *metadata ontology*, called *operation ontology*, used as a template to define Web service operations. A *metadata ontology* provides concepts that allow the description of other concepts (operations in our case) [7].

3.1 Describing Web Service Operations

Each operation is an instance of the operation ontology. It is defined by a set of *nonfunctional* and *functional* attributes. *Nonfunctional* (or *qualitative*) attributes include a set of metrics that measure the quality of the operation (e.g., time, availability, and cost). *Functional* attributes describe syntactic and semantic features of an operation. We identify three groups of functional attributes: *syntactic*, *static semantic*, and *dynamic semantic*. *Syntactic* attributes represent the structure of a service operation. An example of syntactic attribute is the list of input and output parameters that define the operation's messages. *Semantic* attributes refer to the meaning of the operation or its messages. We consider two kinds of semantic attributes: *static* and *dynamic* attributes. *Static* semantic attributes describe features that are not related to the execution of the operation. An example of static attribute is the operation's category (i.e., domain of interest). *Dynamic* semantic attributes refer to the way and constraints under which the operation is executed. An example of dynamic attribute is the business logic of the operation, i.e., the results returned by the operation given certain parameters and conditions.

The concept of *vertical ontology* is key for defining the content of static semantic attributes. It captures the knowledge valid for a particular domain (e.g., government, medical) [7]. For example, NAICS can be used for the category of an operation [1]. Service providers may adopt different vertical ontologies to specify the content of a given parameter. We use XML namespaces to prefix business roles with the taxonomy according to which they are defined [1]. The use of different ontologies to describe an attribute requires dealing with the issue of defining mappings between disparate ontologies. This issue is out of the scope of this paper. However, our model can be extended to deal with ontology mapping (e.g., by adopting one of the techniques presented in [6]).

3.2 Composability Stack

The proposed model for composability contains rules organized into four levels (Fig. 2). Each rule CR_{pq} at a level CL_p (p = 0,3) compares a specific feature of services within CL_p. The first level CL_0 compares syntactic attributes such as the number of message parameters (CR_{00}). The second level CL_1 compares static semantic attributes. We define two groups of rules at this level. The first group compares the static semantics of messages. The second group compares the static semantics of operations. The third level CL_2 compares dynamic semantic attributes. The fourth composability level CL_3 focuses on quality of operation attributes. It contains three groups of rules. The first group compares security attributes. The second group checks business attributes. The third group deals with runtime attributes. Our focus in this paper is on *static semantic* and *dynamic semantic* composability. Details about syntactic and qualitative rules can be found in [14].

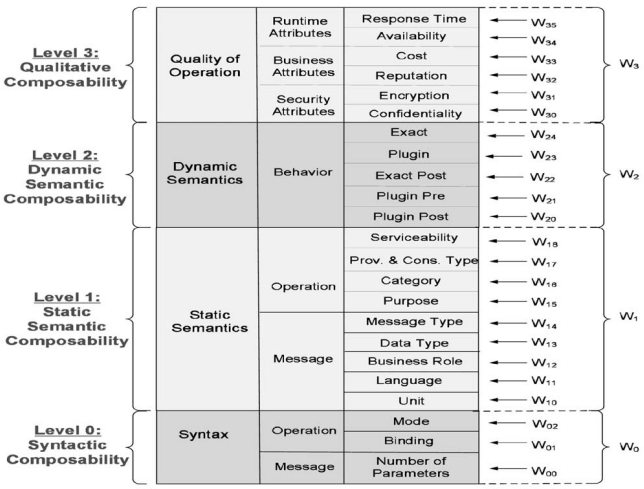


Fig. 2. Web service composability stack.

3.3 Operation Mode and States

Service composition involves the combination of several operations that belong to the same or different Web services. Each operation op_{ik} has an input and output message. Input and output messages contain parameters. The order according to which op_{ik}'s input and output messages are sent and received defines the operation *mode*. The mode indicates whether the operation initiates interactions or simply replies to invocations from other services. We define two modes: *In/Out* or *Out/In*. An *In/Out* operation first receives an input message by a client, processes it, and then returns an output message to the client. *Out/In* first sends an output message to a server and receives an input message as a result. As specified in WSDL standard, some operations may be limited to an input or output message (e.g., notification operation) [1], [14]. Such operations may be considered as *In/Out* or *Out/In* operations where the input or output message is empty.

The execution of an operation op_{ik} generally goes through four major observable states: *Ready*, *Start*, *Active*, and *end*. We define a precedence relationship between states, noted $\longrightarrow_t$, as follows: $S_1 \longrightarrow_t S_2$ if S_1 occurs before S_2. The execution states are totally ordered according to $\longrightarrow_t$ as follows: *Ready* $\longrightarrow_t$ *Start* $\longrightarrow_t$ *Active* $\longrightarrow_t$ *End*. The execution of op_{ik} is in the *Ready* state if the request for executing op_{ik} has not been made yet. The *Start* state means that op_{ik} execution has been initiated. op_{ik} is in the *Active* state if op_{ik} has already been initiated and the corresponding request is being processed. After processing the request, the operation reaches the *End* state during which results are returned.

3.4 Horizontal and Vertical Composition

We define two ways of combining operations: *horizontal* and *vertical*. Each composability rule may be applicable to horizontal composition, vertical composition, or both.

Horizontal composition models a "supply chain" -like combination of operations (Fig. 3). Let op_{ik} and op_{jl} be two operations that are horizontally composed. We call op_{ik} and op_{jl} *source* and *target* operations, respectively. op_{ik} is first executed, followed by op_{jl}'s execution. op_{ik}'s messages are used to *feed* op_{jl}'s input message. Let $\mathcal{M}$ be a set of messages

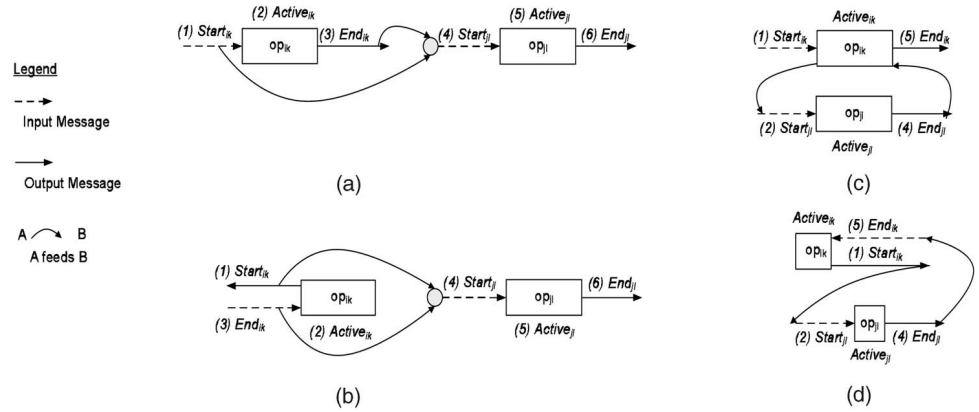


Fig. 3. Horizontal and vertical composition.

and $Input_{jl}$ the input message of op_{jl}. We say that $\mathcal{M}$ *feeds* $Input_{jl}$ if parameters in $\mathcal{M}$'s messages are used as $Input_{jl}$'s parameters. As depicted in Fig. 3, In_{ik} and Out_{ik} messages feed In_{jl}. The precedence relationships between op_{ik}'s and op_{jl}'s states are given below:

- $Start_{ik} \longrightarrow_t Active_{ik} \longrightarrow_t End_{ik}$;
- $End_{ik} \longrightarrow_t Start_{jl} \longrightarrow_t Active_{jl} \longrightarrow_t End_{jl}$.

As example of the horizontal composition (case (a)), assume that op_{ik} provides translation from Chinese to English and op_{jl} provides translation from English to Urdu. The operations op_{ik} and op_{jl} may be horizontally composed to provide translation from Chinese to Urdu. In this case, the output of op_{ik} (English translation) is used as input by op_{jl}. The second case of horizontal composition (case (b)) refers to the situation where op_{ik} outsources from another operation (i.e., op_{ik} is vertically composed with a third operation). op_{ik} is then horizontally composed with op_{jl}. For example, a government agency may have a `get_directions` (Out/In) operation that is executed by outsourcing from other operations `People_Lookup` and `Direction-From-Address` (Fig. 1). The `get_directions` operation is then horizontally composed with `Notify-Citizens` (In/Out operation).

Vertical Composition models the "subcontracting" of an operation op_{jl} by another operation op_{ik} (Fig. 3). Let us consider the first case where op_{ik}'s mode is *In/Out*. Whenever op_{ik} is invoked, it transparently sends an input message to op_{jl}. op_{jl} then performs the requested function on behalf of op_{ik} and returns an output message to op_{ik}. op_{ik} will finally send the results to its invoker. Assume now that op_{ik}'s mode is *Out/In*. op_{ik} starts its execution by invoking op_{jl}. After op_{jl} terminates its execution, it sends results to op_{ik} which receives them as an input message. The precedence relationships between op_{ik}'s and op_{jl}'s are given below:

- $Start_{ik} \longrightarrow_t Start_{jl} \longrightarrow_t Active_{jl} \longrightarrow_t End_{jl} \longrightarrow_t End_{ik}$;
- $Start_{ik} \longrightarrow_t Active_{ik} \longrightarrow_t End_{ik}$.

An example of vertical composition is that of a personal computers (PC) reseller offering an operation `Request-Quotes` (case (c)). This operation allows customers to request quotes. The execution of `Request-Quotes` requires the invocation of another operation provided by a PC manufacturer to get the latest prices. The second case of vertical composition (case (d)) models "request-response" interactions. For example, let us consider an operation `Get-Immunization-Centers` (Out/In) provided by the Department for the Aging. Assume that the list of immunization centers is managed by the Health Department. To get up-to-date information about such centers, it would be more effective to outsource from a corresponding operation in the Health Department (In/Out), whenever `Get-Immunization-Centers` is invoked by a Department of Aging's officer.

3.5 Composability Degree

Composers may have different views on composability rules. One may, for example, give higher importance to syntactic composability while another may focus on semantic rules. To capture this aspect, we associate a *weight* W_p to each level CL_p. We also define a *weight* W_{pq} for each rule CR_{pq} in that level. A weight is an estimate of the significance of the corresponding level or rule from the composer's point of view. Composers assign a weight to each level and rule. The higher the weight, the more important the corresponding level or rule. W_p ($\geq$ 0 and $\leq$ 1) compares CL_p to the other levels in terms of their importance. The total of weights assigned to the different levels equals 1. Similarly, W_{pq} ($\geq$ 0 and $\leq$ 1) compares CR_{pq} to the other rules at level CL_p. The total of weights assigned to rules within a level equals 1. Formally, the different weights must respect the following constraints, where $\mid \mathrm{CL}_p \mid$ is the number of rules at level p:

1. $\forall\, \mathrm{p}, \mathrm{q} \mid 0 \leq \mathrm{p} \leq 3$ and $0 \leq \mathrm{q} \leq \mid \mathrm{CL}_p \mid -1$: $(0 \leq \mathrm{W}_p \leq 1) \wedge (0 \leq \mathrm{W}_{pq} \leq 1)$ and
2. $(\sum_{p=0}^{3} W_p = 1) \wedge (\forall\, \mathrm{p} : \sum_{q=0}^{|\mathrm{CL}_p|-1} W_{pq} = 1)$.

Due to the heterogeneity of Web services, it is not always possible to find operations that are fully composable with source operations. Composers may, in this case, select operations that are partially composable and then adapt their operations based on the results returned by the composability process. For example, the composer may modify the data type of a parameter if it is not compatible with the data type of the corresponding target's parameter. For that purpose, we introduce the notion of composability *degree*.

The *degree* of op_{ik} and op_{jl} gives the ratio of composability rules that are satisfied between op_{ik} and op_{jl}. It takes its values from 0 to 1 ($\geq$ 0 and $\leq$ 1). We define a function $satisfied_{pq}(\mathrm{op}_{ik}, \mathrm{op}_{jl})$ that returns 1 if the rule CR_{pq} is satisfied between op_{ik} and op_{jl} and 0 otherwise. To reflect the composer's view on each rule CR_{pq}, we adjust the value returned by the function $satisfied_{pq}(\mathrm{op}_{ik}, \mathrm{op}_{jl})$ with the weight W_{pq}. The degree at a given level CL_p is obtained by adding the adjusted values returned by the function *satisfied* applied on each CL_p's rule. Once the degree at CL_p is computed, we adjust it with the weight W_p assigned to CL_k. As specified below, the *degree* of op_{ik} and op_{jl} is obtained by summing composability degrees at all levels CL_p (p = 0,4):

$$\mathrm{Degree}\ (\mathrm{op}_{ik}, \mathrm{op}_{jl}) = \sum_{p=0}^{3}(W_p \times \sum_{q=0}^{|\mathrm{CL}_p|-1}(\mathrm{W}_{pq} \times satisfied_{pq}(\mathrm{op}_{ik}, \mathrm{op}_{jl}))).$$

During a composition process, the composer assigns weights to each level and rule by providing a vector called *level weight* (LW) and matrix called *rule weight* (RW). The element LW_p (p = 0,3) gives the weight assigned to level CL_p. The element CW_{pq} gives the weight assigned to rule CR_{pq}. If a rule CR_{pq} is undefined, then CW_{pq} is automatically assigned the value 0. Additionally, if the weight of a given level is equal to 0, then the weight of each rule within that level is also equal to 0.

Based on the degree of op_{ik} and op_{jl}, we can decide about the composability of those operations. If *degree* = 0, then no rule is satisfied and the operations are *noncomposable*. If *degree* = 1, then all composability rules (with a positive level and rule weight) are satisfied and the operations are *fully* composable. Otherwise, a subset of rules are satisfied. In this case, op_{ik} and op_{jl} are *partially* composable.

3.6 τ-Composability

Composers may have different expectations about the composability degree of their operations. For that purpose, they provide a *composability threshold* $\tau\ (0 < \tau \leq 1)$ which gives the minimum value allowed for a composability degree. All operations op_{jl} so that $degree(\mathrm{op}_{ik}, \mathrm{op}_{jl}) \geq \tau$ are candidates to be composed with op_{ik}. If the threshold is greater than $\mathrm{degree}(\mathrm{op}_{ik}, \mathrm{op}_{jl})$, then op_{ik} is not composable with op_{jl}. Based on the notions of degree and threshold, we introduce a "relaxed" definition of composability called *τ-composability*. *τ-composability* compares the composability degree and threshold to decide whether an operation is composable with another from the composers' perspectives. op_{ik} is *τ-composable* with op_{jl} if $degree(\mathrm{op}_{ik}, \mathrm{op}_{jl}) \geq \tau$.

The composability threshold is given by composers as part of their *profile*. Composers personalize the composability checking process via their profile. They assign values to the level weights vector (LW), rule weights matrix (RW), and τ. Other variables such as the maximum number of target operations can also be initialized. The way users create their profile depends on their level of expertise. We identify three types of users: *casual* (i.e., with minimal expertise), *expert* (i.e., with high expertise), and *regular* (i.e., with average expertise). *Casual* users may leave LW and RW unassigned in their profile. The system automatically distributes weights between levels and rules in a uniform way. The composability threshold will also be set to 1. In this case, full composability will be required. *Expert* users are knowledgable about the meaning of all operation and message attributes. They may customize the composability process by assigning the desired values to LW, RW, and τ. If the degree exceeds the threshold but is not equal to 1, users change the specification of their operations based on the feedback returned by the system (e.g., which rules are not satisfied) to increase the degree. The third type of users, called *regular* users, includes those that have *some* knowledge about operation and message attributes. They may assign values to parts of LW and RW. In this case, the system automatically distributes weights between unassigned levels and rules. If τ was not assigned by a user, it is automatically set to 1 by the system.

3.7 Interoperation Relationships

Executing an operation may require going through a predefined process (called *behavior*) that involves the execution of other operations. We define two types of relationships between operations: *preoperations* and *post operations*. These relationships may be dictated by government regulations. For example, senior citizens must first register with the agency via `checkRegistration` operation before applying for any welfare program. They may also reflect the business logic of a Web service. For example, senior citizens must order a meal from a participating restaurant via the `orderMeal` operation before requesting its delivery through the `mealsOnWheels` operation. Pre and postoperations are defined as follows:

- *Preoperations*. op_{ik} is a *preoperation* of op_{jl} if the invocation of op_{jl} is preceded by the execution of op_{ik} (i.e., $End(\mathrm{op}_{ik}) \longrightarrow_t Ready(\mathrm{op}_{jl})$). An operation may have several preoperations and be the preoperation of several operations. An operation is invoked only if all its preoperations have reached their "End" state.
- *Postoperations*. The execution of a given operation may trigger the invocation of other operations called *postoperations*. op_{ik} is a *postoperation* of op_{jl} if the termination of op_{jl} precedes the invocation of op_{ik} (i.e., $End(\mathrm{op}_{jl}) \longrightarrow_t Ready(\mathrm{op}_{ik})$). An operation may have several postoperations. It may also be the postoperation of several operations. A postoperation enters the "Start" state if *at least* one of its sources is in the "End" state.

Pre and postoperations introduce constraints that must be considered during the composability checking process [8]. Let us consider `orderMeal` and `mealsOnWheels` operations defined previously. Assume that an operation op_1 is found to be vertically composable with `mealOnWheels` (i.e., op_1 can outsource from `mealsOnWheels`). Since `orderMeal` is a preoperation of `mealsOnWheels`, op_1 cannot outsource from `mealsOnWheels` unless the system makes sure that `orderMeal` or another operation composable with `orderMeal` is invoked before op_1. The focus in this paper is on operations that are independent from each other.

4 Semantic Description of Web Services

In this section, we present our framework for the semantic description of service operations. This framework is used as a foundation for checking composability. In our approach, operations are semantically described at two levels: *static* and *dynamic*. The *static semantics* of an operation models "noncomputational" properties of an operation, that is, properties that are independent of the execution of the operation. The static semantics is described at two "granularities" : operation and messages. The *dynamic semantics* of an operation models computational or execution-related features of that operation. It generally refers to the way and constraints under which an operation is executed.

Significant research is being devoted to the definition of a service ontology in *DAML-S* [12]. The approach presented in this section can be combined with DAML-S to provide richer semantic features. Indeed, the static semantics in DAML-S mostly focuses on describing operations' features. We define a broader view of static semantics by describing semantics at both the operation and message levels. Additionally, DAML-S gives little support for dynamic semantics.

4.1 Static Semantics of Operations

The static semantics at the operation granularity is defined by the following attributes:

- *Serviceability*. This attribute gives the type of assistance provided by the operation. Examples of values for this attribute are "cash" and "in-kind." TANF (Temporary Assistance for Needy Families) is an example of service that provides financial support to needy families.
- *Provider and consumer types*. The provider of an operation may be governmental ("federal," "state," "local," etc.) or nongovernmental ("nonprofit" and "business") agencies. For example, `nursingHome` may be provided by the Department of Aging (government) and Red Cross (nonprofit). The *consumer type* specifies the group of citizens (e.g., children, pregnant women) that are eligible to the operation's welfare program. For example, WIC (Women, Infant, and Children) is a program for pregnant women, lactating mothers, and children.
- *Category*. The *category*C_{ik} of an operation op_{ik} describes the area of interest of op_{ik}. It is defined by a tuple (Dom_{ik}, Syn_{ik}, $Spec_{ik}$, $Overlap_{ik}$). Dom_{ik} gives the area of interest of the community (e.g., "healthcare"). It takes its value from a vertical ontology for domain names. Syn_{ik} contains a set of alternative domain names for C_{ik}. For example, "medical" is a synonym of "healthcare." $Spec_{ik}$ is a set of specializations of C_{ik}'s domain. For example, "insurance" and "children" are specializations of "healthcare." This means that C_{ik} provides health insurance services for children. $Overlap_{ik}$ contains the list of categories that *overlap* with C_{ik}'s category. It is used to provide a peer-to-peer topology for connecting operations with "related" categories. We say that $category_{ik}$ *overlaps with* $category_{jl}$ if composing op_{ik} with op_j's is "meaningful." By meaningful, we mean that the composition provides a *value-added* service (in terms of categories). For example, an operation that have *family* as a domain may be composed with another operation whose domain is *insurance*. This would enable providing health insurance for needy families.
- *Purpose*. The *purpose* describes the goal of the operation. It is defined by four attributes: *Func*, *Syn*, *Spec*, and *Overlap*. The *Func* describes the business functionality offered by the operation. Examples of functions are "eligibility," "registration," and "mentoring." The *Syn*, *Spec*, and *Overlap* attributes work as they do for categories. The *Overlap* contains the list of purposes that are related to the purpose of the current operation. For example, two operations that have "eligibility" and "registration" as respective purposes may be combined to first check whether citizens are eligible for a given social a program and then register them for that program.

4.2 Static Semantics of Messages

Each message within an operation is semantically described via a *message type* $\mathcal{MT}$. $\mathcal{MT}$ gives the general semantics of the message. For example, a message may represent a "purchase order" or an "invoice."

Message types do not capture the semantics of parameters within a message. We define below a set of attributes to model the semantics of message parameters:

- *Data type*. It gives the range of values that may be assigned to the parameter. We use XML Schema's *built-in* data types as the typing system. *Built-in* (or simple) types are predefined in the XML Schema specification. They can be either *primitive* or *derived*. Unlike *primitive* types, *derived* types are defined in terms of other types. For example, *integer* is derived from the *decimal* primitive type. Complex data types can also be adopted in our model but are out of the scope of this paper [9].
- *Business role*. It gives the type of information conveyed by the message parameter. For example, an `address` parameter may refer to the first (street address and unit number) or second (city and zip code) line of an address. Business roles take their values from a predefined taxonomy. Every parameter would have a well-defined meaning according to that taxonomy. An example of such taxonomy is *RosettaNet*'s business dictionary [13]. It contains a common vocabulary that can be used to describe business properties.
- *Unit*. It refers to the measurement unit in which the parameter's content is provided. For example, a *weight* parameter may be expressed in "Kilograms" or "Pounds." An *eligibility period* parameter may be specified in days, weeks, or months. We use standard measurement units (length, area, weight, money code, etc.) to assign values to parameters' units. If a parameter does not have a unit (e.g., *address*), its unit is equal to "none."

TABLE 1
Static Semantic Rules for Operations

CR(op_{ik}, op_{jl})	*Vertical Composition*	*Horizontal Composition*
Serviceability	• $\text{serviceability}_{ik} = \text{serviceability}_{jl}$	• Not Applicable
Provider/Consumer	• $\text{provider}_{ik} \cap \text{provider}_{jl} \neq \emptyset$ and $\text{consumer}_{ik} \cap \text{consumer}_{jl} \neq \emptyset$	• $\text{provider}_{ik} \cap \text{consumer}_{jl} \neq \emptyset$
Category	• $\text{Spec}_{ik} \subseteq \text{Spec}_{jl}$; and • $(\text{Dom}_{ik} = \text{Dom}_{jl})$ or $(\text{Dom}_{ik} \in \text{Syn}_{jl})$ or $(\text{Dom}_{jl} \in \text{Syn}_{ik})$ or $(\text{Syn}_{ik} \cap \text{Syn}_{jl} \neq \emptyset)$	• $\text{category}_{jl} \in \text{Overlap}_{ik}$
Purpose	• $\text{Spec}_{ik} \subseteq \text{Spec}_{jl}$; and • $(\text{Func}_{ik} = \text{Func}_{jl})$ or $(\text{Func}_{ik} \in \text{Syn}_{jl})$ or $(\text{Func}_{jl} \in \text{Syn}_{ik})$ or $(\text{Syn}_{ik} \cap \text{Syn}_{jl} \neq \emptyset)$	• $\text{purpose}_{jl} \in \text{Overlap}_{ik}$

- *Language*. The content of a message parameter may be specified in different languages. For example, an `English-Urdu-translation` operation takes as input an English word and returns as output its translation in Urdu. We adopt the standard taxonomy for languages to specify the value of this attribute.

4.3 Dynamic Semantics

The *dynamic semantics* or *business logic* of an operation op_{ik} refers to the outcome expected after executing op_{ik} given a specific condition. Service providers may decide beforehand which "effects" are made visible to users. In the case of e-government, for example, the rules for citizens' eligibility are "public." A `check_eligibility` operation for a given social program returns whether citizens are eligible or not given certain conditions.

The business logic of an operation is defined by a set of *rules* where each rule R_{ik}^m has the following format:

$$R_{ik}^m = \frac{(PreParameters_{ik}^m, PreCondition_{ik}^m)}{(PostParameters_{ik}^m, PostCondition_{ik}^m)}.$$

$\text{PreParameters}_{ik}^m$ and $\text{PostParameters}_{ik}^m$ are sets of parameters. Each parameter is defined by name, data type, business role, unit, and language as stated in Section 4.2. The elements of $\text{PreParameters}_{ik}^m$ and $\text{PostParameters}_{ik}^m$ generally refer to op_{ik}'s input and output parameters. However, they may in some cases refer to parameters that are neither input nor output of op_{ik}. For example, assume that the *address* of every citizen registered with the Department on the Aging is stored in the department's database. In this case, this parameter should not be required as input for the `orderMeal` operation since its value could be retrieved from the database.

$\text{PreCondition}_{ik}^m$ and $\text{PostCondition}_{ik}^m$ are conditions over the parameters in $\text{PreParameters}_{ik}^m$ and $\text{PostParameters}_{ik}^m$, respectively. They are specified as predicates in first-order logic. The rule R_{ik}^m specifies that if $\text{PreCondition}_{ik}^m$ holds when the operation op_{ik} starts, then $\text{PostCondition}_{ik}^m$ holds after op_{ik} reaches its End state. If $\text{PreCondition}_{ik}^m$ does not hold, there are no guarantees about the outcome of the operation. The following is an example of the pre and postcondition of a rule associated with the operation `registerFoodCheck`:

$$\frac{income < 22,090 \wedge size \geq 2 \wedge zip = 22,044}{approved = true \wedge duration = 6},$$

The rule uses *income* (unit = {year, US dollar}), *familySize*, *zip*, *approved*, and *duration* (unit = {month}) as parameters. It states that citizens with a yearly income less than 22,090 US dollars, a minimum household size 2, and living in area code 22,044 are eligible for food checks for a 6-month period.

5 Semantic Composability

For two operations op_{ik} and op_{jl} to be "plugged" together, they must be semantically "compliant." In this section, we present composability rules at the static and dynamic semantic levels. We define each rule with regard to vertical and horizontal composability. For the static semantics, we consider composability at both operation and message granularities.

5.1 Static Semantic Composability of Operations

We summarize in Table 1 the static semantic rules at the operation granularity. The first rule compares op_{ik}'s and op_{jl}'s *serviceability*. Horizontal composition does not require comparing *serviceability* since no operation will "service" the other. However, the content of both attributes must be "similar" if op_{ik} and op_{jl} are vertically composed

TABLE 2
Static Semantic Rules for Messages

CR(op_{ik}, op_{jl})	*Vertical Composition and Mode$_{ik}$ = In/Out*	*Vertical Composition and Mode$_{ik}$ = Out/In*	*Horizontal Composition*
Message Type	• $\mathcal{MT}(\text{In}_{ik}) = \mathcal{MT}(\text{In}_{jl})$; and • $\mathcal{MT}(\text{Out}_{ik}) = \mathcal{MT}(\text{Out}_{jl})$	• $\mathcal{MT}(\text{Out}_{ik}) = \mathcal{MT}(\text{In}_{jl})$; and • $\mathcal{MT}(\text{Out}_{jl}) = \mathcal{MT}(\text{In}_{ik})$	• Not Applicable
Data Type	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \mid \text{compatible}(p',p)$; and • $\forall p \in \text{Out}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{compatible}(p',p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{Out}_{ik} \mid \text{compatible}(p',p)$; and • $\forall p \in \text{In}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{compatible}(p',p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid \text{compatible}(p',p)$
Role	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \mid \text{role}(p') = \text{role}(p)$; and • $\forall p \in \text{Out}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{role}(p') = \text{role}(p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{Out}_{ik} \mid \text{role}(p') = \text{role}(p)$; and • $\forall p \in \text{In}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{role}(p') = \text{role}(p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid \text{role}(p') = \text{role}(p)$
Language	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \mid \text{lang}(p') = \text{lang}(p)$; and • $\forall p \in \text{Out}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{lang}(p') = \text{lang}(p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{Out}_{ik} \mid \text{lang}(p') = \text{lang}(p)$; and • $\forall p \in \text{In}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{lang}(p') = \text{lang}(p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid \text{lang}(p') = \text{lang}(p)$
Unit	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \mid \text{unit}(p') = \text{unit}(p)$; and • $\forall p \in \text{Out}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{unit}(p') = \text{unit}(p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{Out}_{ik} \mid \text{unit}(p') = \text{unit}(p)$; and • $\forall p \in \text{In}_{ik}\ \exists p' \in \text{Out}_{jl} \mid \text{unit}(p') = \text{unit}(p)$.	• $\forall p \in \text{In}_{jl}\ \exists p' \in \text{In}_{ik} \cup \text{Out}_{ik} \mid \text{unit}(p') = \text{unit}(p)$

($\text{serviceability}_{ik} = \text{serviceability}_{jl}$). For example, an operation providing cash support cannot "subcontract" in-kind operations. The "=" operator used here refers to one of the following two cases: 1) $\text{serviceability}_{ik}$ and $\text{serviceability}_{jl}$ have the same content if the attributes use the same vertical ontology; or 2) $\text{serviceability}_{ik}$ and $\text{serviceability}_{jl}$ have "mappable" content if the attributes use different vertical ontologies.

The second rule compares op_{ik}'s and op_{jl}'s *provider* and *consumer types*. If the composition is vertical, op_{ik} and op_{jl} must have at least one common provider type and one common consumer type. For example, if op_{ik} expects to outsource from a federal agency's operation, then op_{jl}'s agency should include the type "federal." Additionally, if op_{ik} provides benefits for children and pregnant women, then op_{jl} should provide benefits for at least those two groups. If op_{ik} is horizontally composed with op_{jl}, then it should be viewed as a consumer of op_{jl}. Hence, op_{jl}'s consumer type should include at least one value from op_{ik}'s provider types.

The third rule compares operations' *categories*. Assume that op_{ik} is vertically composed with op_{jl}. Since op_{ik} is meant to "replace" op_{jl}, the following two conditions should be true: 1) op_{ik}'s and op_{jl}'s domains of interest are similar or synonyms and 2) all characteristics (i.e., elements of the "spec" attribute) of op_{ik}'s category are provided by op_{jl}'s. For example, assume that op_{ik}'s category provides health insurance for children (i.e., $\text{Dom}_{ik} = "healthcare"$ and $\text{Spec}_{ik} = \{"children," "insurance"\}$). The operation op_{jl} should not only deal with healthcare but also *at least* provide insurance for children as well. Assume now that op_{ik} is horizontally composed with op_{jl}. Category_{ik} and category_{jl} should be defined so that op_{ik} and op_{jl} "can" be combined. This is captured by the *Overlap* attribute of a category. Hence, category_{ik} is composable with category_{jl} if Overlap_{ik} contains category_{jl}.

The last rule compares operations' *purposes*. The *purpose composability* rule is defined in the same way as *category composability* where Dom is replaced by Func.

5.2 Static Semantic Composability for Messages

Table 2 summarizes the static semantic rules at the message granularity. The first rule compares op_{ik}'s and op_{jl}'s message types. This rule is applicable only to vertical composition since horizontal composition does not involve replacing op_{ik}'s messages with op_{jl}'s or vice versa. Assume

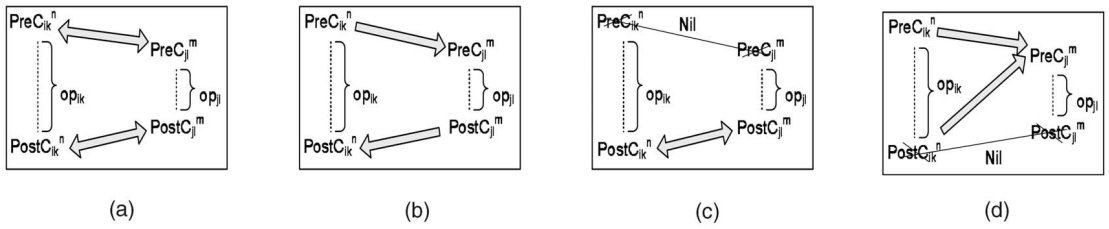


Fig. 4. B-composability rules. (a) Exact b-composability. (b) Plugin b-composability. (c) Exact post-b-composability. (d) plugin pre-b-composability.

that op_{ik} is vertically composed with op_{jl}. As depicted in Fig. 3, we identify two cases based on the mode of op_{ik}. If op_{ik}'s mode is *In/Out*, then In_{ik}'s (respectively, Out_{ik}'s) and In_{jl}'s (Out_{jl}'s) types should be similar. If op_{ik}'s mode is *Out/In*, then Out_{ik}'s (respectively, In_{ik}'s) and In_{jl}'s (Out_{jl}'s) types should be similar.

The second composability rule compares parameters' data types. It is based on the notion of *compatibility* between data types (XML Schema). Two parameters are *data type compatible* if they have the same built-in type. Compatibility of derived types and complex data types can also be adopted. However, these issues are out of the scope of this paper. A discussion about typing in XML can be found in [9].

Data type composability depends on the composition type (horizontal or vertical) and operations' modes. As depicted in Fig. 3, we identify the following four cases: If op_{ik} is vertically composed with op_{jl} and $Mode_{ik} = "In/Out"$, then In_{ik} is "plugged" with In_{jl} and Out_{jl} is "plugged" with Out_{ik} (Fig. 3c). The data type of each parameter in $In_{jl}(Out_{ik})$ should be compatible with the data type of a corresponding parameter in $In_{ik}(Out_{jl})$. If op_{ik} is vertically composed with op_{jl} and $Mode_{ik} = "Out/In"$, then Out_{ik} is "plugged" with In_{jl} and Out_{jl} is "plugged" with In_{ik} (Fig. 3d). The data type of each parameter in In_{jl} (respectively, In_{ik}) should be compatible with the data type of a corresponding parameter in Out_{ik} (Out_{jl}). If op_{ik} is horizontally composed with op_{jl}, then In_{ik} and Out_{ik} are "plugged" with In_{jl} independently of op_{ik}'s mode (Figs. 3a and 3b). The data type of each parameter in In_{jl} should be compatible with the data type of a corresponding parameter in In_{ik} or Out_{ik}.

The remaining three rules compare parameters' *business role*, *language*, and *unit*, respectively. They are defined similarly to data type composability, except that the data type is replaced by *business role*, *language*, and *unit*, respectively.

5.3 Dynamic Semantic Composability

The *dynamic semantic* composability (or *B-Composability*) compares the business logic rules of source and target operations. Let us consider two rules

$$R^n_{ik} = (PreC^n_{ik}, PostC^n_{ik})$$

and $R^m_{jl} = (PreC^m_{jl}, PostC^m_{jl})$ that belong to op_i and op_j, respectively. B-composability relates $PreC^n_{ik}$ to $PreC^m_{jl}$ and $PostC^n_{ik}$ to $PostC^m_{jl}$. We define several forms of B-composability depending on the relationships between post and preconditions. Each form is an instantiation of the general form of B-composability, called *generic B-composability*. We say that op_{ik} is *Generically B-composable* with op_{jl} if:

$$\forall\ R^n_{ik} \in Rules(\mathrm{op}_{ik})\ \exists\ R^m_{jl} \in Rules(\mathrm{op}_{jl})\ |$$
$$(\widetilde{PreC}_{ikn}\ \mathcal{R}_1\ PreC^m_{jl})\ \wedge\ (PostC^m_{jl}\ \mathcal{R}_2\ PostC^n_{ik}).$$

The relations $\mathcal{R}_1$ and $\mathcal{R}_2$ relate preconditions and post conditions, respectively. Each relation is either equivalence ($\Leftrightarrow$), implication ($\Rightarrow$), or *nil* (meaning that the corresponding term is dropped). As illustrated in this section, we may need to include information about the postcondition in the precondition clause. To allow this flexibility, we define $\widetilde{PreC}_{ikn}$ as either $PreC^n_{ik}$ or $PreC^n_{ik} \wedge PostC^n_{ik}$ in the generic B-composability rule. Note that techniques for comparing pre and postconditions have been presented in [21]. However, these techniques deal with component-based environments *not* Web services.

Fig. 4 depicts the different forms of B-Composability rules. We first give the strongest rule and then weaken the rules by relaxing $\mathcal{R}_1$ and $\mathcal{R}_2$ from $\Leftrightarrow$ to $\Rightarrow$, and *nil*. We also vary $\widetilde{PreC}_{ikn}$ from $PreC^n_{ik}$ to $PreC^n_{ik} \wedge PostC^n_{ik}$. Relaxing the rules enables the comparison of less closely related operations:

- *Exact*. Exact B-composability instantiates $\mathcal{R}_1$ and $\mathcal{R}_2$ to $\Leftrightarrow$ and $\widetilde{PreC}_{ikn}$ to $PreC^n_{ik}$ (Fig. 4a). If two operations are exactly B-composable, then their business logics are equivalent. Hence, whenever one operation is used, it could be replaced by the other with no change in observable business logic. This rule is suitable for vertical composition since op_{ik} and op_{jl} are in their *active* state simultaneously.
- *Plugin*. This rule relaxes both $\mathcal{R}_1$ and $\mathcal{R}_2$ from $\Leftrightarrow$ to $\Rightarrow$. It also instantiates $\widetilde{PreC}_{ikn}$ to $PreC^n_{ik}$. The rule R^n_{ik} is matched by any rule R^m_{jl} whose precondition is weaker to allow *at least* all of the conditions that R^n_{ik} allows. The postcondition of R^m_{jl} is stronger than R^n_{ik}'s to provide a condition *at least* as strong as R^n_{ik}'s. As depicted in Fig. 4b, this rule is suitable for vertical composition since op_{ik} and op_{jl} are in their *active* state simultaneously.
- *Exact Post*. In some cases, composers are concerned only with the effects of operations. For example, a composer may be interested in an operation that provides a social benefit independently of any precondition of that operation. Thus, a useful relaxation of the exact B-composability is to consider only the postcondition part of the conjunction. *Exact post* is also an instance of the generic B-composability, with $\mathcal{R}_2$ instantiated to $\Leftrightarrow$ and dropping both $\widetilde{PreC}_{ikn}$ and $PreC^m_{jl}$ (Fig. 4c). Since only the equivalence relationship is used, the exact post is symmetric. Because op_{ik}

and op_{jl} are in their *active* state simultaneously (Fig. 4c), this rule is suitable for vertical composition.

- *Plugin Pre*. Plugin Pre includes information about op_i's postcondition in the precondition and drops the relationship between postconditions. It is an instantiation of generic B-composability where $\mathcal{R}_1$ is instantiated to $\Rightarrow$, $\mathcal{R}_2$ to *nil*, and $\widetilde{PreC}_{ikn}$ to $PreC^n_{ik} \wedge PostC^n_{ik}$. This rule is particularly useful to check horizontal composability, that is, whether the execution of op_i can be followed by the execution of op_j. Fig. 4d shows that op_{ik} and op_{jl} enter their *active* states sequentially ($Active_{ik} \longrightarrow_t Active_{jl}$). Since op_i is executed (according to R^n_{ik}) before op_j, $PreC^n_{ik}$, and $PostC_{kl}$ are by definition true. In order for op_j to be executable according to R^m_{jl}, its precondition $PreC^m_{jl}$ should be true. One way to ensure this is to check that the implication $PreC^n_{ik} \wedge PostC^n_{ik} \Rightarrow PreC^m_{jl}$ is true.

6 Checking Service Composability

We identify three avenues in the area of service composition that could benefit from checking composability: composition analysis, automatic composition, and operation outsourcing. In the first case, composability is checked a posteriori. Composers first specify their composite service (e.g., using BPEL4WS). The composition engine then checks the "correctness" of the composite service using composability rules. In the second case, the composition engine uses the composability rules to generate composite service descriptions from high-level specifications of composition requests. The engine needs to determine the set of participants relevant to the composition request while making sure that they are composable. In the third case, composability rules are used to "replace" an operation by a "compatible" one. This could be useful to enable the subcontracting of operations or substitutability of a participant by another. In this section, we propose an algorithm for checking composability in the case of operation outsourcing.

6.1 Algorithms

The aim of our algorithm (Fig. 5) is to determine the set of all operations within the service registry that could be outsourced by a given source operation *SO*. The target operation should be "similar enough" to the source operation so that it could be invoked instead of *SO*. We formulate this problem using our composability model as follows: "determine the set $\mathcal{T}$ of target operations *op* within the registry so that *SO* is vertically composable with *op*." The degree of similarity between *SO* and *op* is defined by τ.

The algorithm browses the registry, checking the vertical composability of *SO* with every operation *op* in the registry (Fig. 5). As the number of target operations may be large, users have the possibility to set in their profile the maximum number of target operations to be determined (*max_target* variable). Composability degree is computed after checking composability at each level and granularity. If the degree is greater or equal to τ, then *op* is a potential candidate to "replace" *SO*. In this case, *op* is added to $\mathcal{T}$. Users will be able to select the "best" operations to be outsourced via qualitative composability. Details about qualitative composability are outside the scope of this paper.

```
Algorithm: Composability_Checking
Input: SO, Registry, T, Profile, nb_target
Output: List T of operations vertically composable with SO
(01) nb_target = 0;  degree = 0;  T = ∅;
(02)  for each operation op ∈ Registry |
(03)          (nb_target < max_target) do {
(04)   static_semantics_operation(SO, op, degree)
(05)   If degree > τ Then { T = T ∪ {op};
(06)                        nb_target = nb_target + 1;
(07)                        Continue; }
(08)   static_semantics_message(SO, op, degree)
(09)   If degree > τ Then { T = T ∪ {op};
(10)                        nb_target = nb_target + 1;
(11)                        Continue; }
(12)   dynamic_semantics(SO, op, degree)
(13)   If degree > τ Then { T = T ∪ {op};
(14)                        nb_target = nb_target + 1;
(15)                        Continue; }  }
```

Fig. 5. Composability algorithm.

The static semantic procedures are a straightforward implementation of vertical composability rules as defined in Tables 1 and 2, respectively. The *dynamic_semantics()* procedure (Fig. 6) compares *SO*'s business logic rules with *op*'s. Since B-composability rules are hierarchically organized, we adopt a bottom-up approach for checking these rules. For example, if the *Plugin Pre* rule is not satisfied, then the *Plugin* rule is necessarily not satisfied since *Plugin* $\Rightarrow$ *Plugin Pre*. A rule is checked if the corresponding weight is positive. For that purpose, we use a function *get_ weightR(CR)* that returns the weight of the current rule *CR* from the RW matrix. We also use a function *get_weightL(CL)* that returns the weight of the current level *CL* from the LW vector.

B-composability rules are mainly based on proving implications between conditions (pre and postconditions) within a pair of business logic rules. However, proving theorems such as $Cond_1 \Rightarrow Cond_2$ is a NP-complete problem [16]. To deal with this issue, we define an approximate solution for proving such theorems. The proposed solution (Fig. 7) is based on the assumption that each condition is a conjunction of *terms*. Each *term* has the form $x < \mathbf{r} > v$ where x is parameter, v is a constant value, and $< \mathbf{r} >$ is a relational operator that belongs to $\{=, \neq, <, >, \leq, \geq\}$.

The theorem prover (Fig. 7) first *unifies* the parameters in $Cond_1$ and $Cond_2$. The unification step works as follows: If there are parameters $x^1_1, \ldots, x^1_n$ in $Cond_1$ that are composable with parameters $x^2_1, \ldots, x^2_m$ in $Cond_2$, then replace $x^1_1, \ldots, x^1_n$ and $x^2_1, \ldots, x^2_m$ by the same parameter name (say

Algorithm: *Dynamic_Semantics*

Input: *op, op', degree*

Output: *degree, are B-composability rules satisfied or not?*

```
(01) d = 0;  /* d = degree for dynamic semantic composability */
(02)         /* degree = total composability degree */
(03) isExact = isExactPost = isPluginPre = isPlugin = false;
(04) if get_weightR(CR) > 0
(05)   then isPluginPre = check_plugin_pre(op,op');
(06) if ¬isPluginPre
(07)   then isPlugin = false
(08)   else if get_weightR(CR) > 0
(09)             then isPlugin = check_plugin(op,op');
(10) if get_weightR(CR) > 0
(11)   then isExactPost = check_exact_post(op,op');
(12) if ¬isPlugin or ¬isExactPost
(13)   then isExact = false
(14)   else if get_weightR(CR) > 0
(15)             then isExact = check_exact(op,op');
(16) if isPluginPre then d = d + get_weightR(CR);
(17) if isPlugin then d = d + get_weightR(CR);
(18) if isExactPost then d = d + get_weightR(CR);
(19) if isExact then d = d + get_weightR(CR);
(20) degree = degree + d × get_weightL(CL);
```

Fig. 6. Business logic composability.

x_1^2). Composability between parameters refers to the verification of composability rules between message parameters. The second step of the prover is to *match* each term t_{2p} of $Cond_2$ with a term t_{1q} in $Cond_1$. We say that t_{2p} *matches* with t_{1q} if $t_{1q} \Rightarrow t_{2p}$. Proving the matching between terms is done by applying one of the *inference rules* for relational operators. Note that the number of inference rules is finite. If a given term in $Cond_2$ matches with no term in $Cond_1$, then $Cond_1 \Rightarrow Cond_2$ is false. If all terms in $Cond_2$ are matched with a term in $Cond_1$, then $Cond_1 \Rightarrow Cond_2$ is true. As an example, we give below the inference rules for the ">" operator:

- $\frac{x>a \wedge a=b}{x>b}$; $\frac{x>a \wedge a>b}{x>b}$; $\frac{x>a \wedge a\geq b}{x\geq b}$; $\frac{x>b}{x\geq b}$.

The composability algorithms return a Boolean answer for whether operations are τ-composable (by comparing τ and the computed degree). However, a log containing details about which composability rules were not satisfied can also be returned. For example, a data type incompatibility between message parameters can be mentioned in the log by the `static_semantics_message` procedure. Composers have then the possibility to review that log and, if possible, modify the description of their outsourcing requests to increase the composability degree of their requests. For the sake of simplicity, we did not include such details in our algorithms.

Algorithm: *Prover*

Input: $Cond_1$, $Cond_2$

Output: *Test whether* $\text{Cond}_1 \Rightarrow \text{Cond}_2$

```
(01) while true do
(02)  If ∃ x_1^1,...,x_n^1 in Cond_1 composable with x_1^2,...,x_m^2 in Cond_2
(03)    then replace x_1^1,...,x_n^1 and x_1^2,...,x_m^2 by x_1^2
(04)    else break
(05) for each term t_2p in Cond_2 do
(06)  If ∃ term t_1q in Cond_1 | t_1q ⇒ t_2p
(07)    then return true
(08)    else return false
```

Fig. 7. Theorem power.

6.2 Analytical Model

In this section, we present the analytical model for the composability algorithm. We focus on computing the total time for checking composability of all target operations. This time will be the sum of the global time for checking static (T_{SS}) and dynamic (T_{DS}) semantic composability. Table 3 defines the parameters and symbols used in this section.

We compute the average execution time for the algorithm. Thus, T is equal to $(T_{min} + T_{max})/2$. To simplify the analysis, we assume that the times to retrieve a description from a service registry and parse that description are fixed values. It is also reasonable to assume that time to check static semantic (at the operation granularity) composability for an operation is a constant. In contrast, message and business logic composability times depend on the number of message parameters and business logic rules, respectively. Thereby, U_t, O_t, t_{ST}, and t_{SS} are constants.

Let us start by computing the minimum composability time T_{min}. This time corresponds to the case where N_T iterations of the algorithm are executed. This means that we get a composability plan after each iteration. The total static semantic composability time T_{SS}^{min} is equal to $N_T \times (t_{SS} + T_{msg})$. Let us now compute the time T_{msg} for checking message composability. T_{msg} refers to the time of comparing a pair of message twice. At minimum, each parameter in a message would be compared to one parameter of the dual message. Hence, T_{msg} is equal to $2 \times P_M$ and T_{SS}^{min} is equal to $N_T \times (t_{SS} + 2 \times P_M)$. The last time to compute is T_{DS}^{min}. Since the algorithm performs vertical composability, dynamic semantic composability does not check plugin prerule. Based on the definition of B-composability rules, there is a need to execute the theorem prover eight times: two for exact post, two for plugin, and four for exact B-composability. At minimum, each business rule would be proven using the first inference rule. T_{DS}^{min} is then equal to $N_T \times (8 \times B_{op} \times TC)$. Based on the above analysis, we have:

$$T_{min} = N_T \times (U_t + O_t + t_{SS} + 2 \times P_M + 8 \times B_{op} \times TC).$$

TABLE 3
Symbols and Parameters

Variables	
N_{op}	Number of operations in the registry
N_T	Maximum number of target operations
n_{SO}	Number of source operations
P_M	Number of parameters per message
B_{op}	Number of business rules per operations
TC	Number of terms per condition
I_R	Number of inference rules
Performance measurement parameters and functions	
U_t	Time to obtain description from service registry
O_t	Time to parse a service description
T_{SS}^{min}	Minimum total static semantic composability time
T_{DS}^{min}	Minimum total dynamic semantic composability time
T_{SS}^{max}	Maximum total static semantic composability time
T_{DS}^{max}	Maximum total dynamic semantic composability time
t_{SS}	Time to check static composability (operation granularity)

The maximum composability time T_{min} refers to the case where all operations in the registry are checked. This means that the number of iterations executed by the algorithm is equal to N_{op}. Consequently, T_{SS}^{max} is equal to $N_{op} \times (t_{SS} + T_{msg})$. Let us now compute the formula for T_{msg}. The first parameter of each message should be compared to P_M parameters of the other message. The next parameter needs to be compared to $P_M - 1$ parameters, and so on. Hence, T_{msg} is equal to

$$2 \times (P_M + (P_M - 1) + ... + 1) = P_M \times (P_M + 1).$$

Now, we need to compute T_{DS}^{max}. As mentioned previously, we execute the theorem prover eight times to check the different B-composability rules. We compare each rule in the subrequest with all rules of the current operations. The total of comparisons is then B_{op}^2. For each term, we need to go through all inference rules for a total of $TC \times I_R$. T_{DS}^{max} is then equal to $N_{op} \times (8 \times B_{op}^2 \times TC \times I_R)$. The maximum composability time is then given below:

$$T_{max} = N_{op} \times (U_t + O_t + t_{SS} + P_M \times (P_M + 1) + 8 \times B_{op}^2 \times TC \times I_R).$$

The previous formulas give composability times for one outsourcing request. In what follows, we specify the total execution time for the composability algorithm when executed on n_{SO} source operations. In this case, we consider the times to access the registry and parse operation descriptions:

$$T = \frac{1}{2} \times N_{op} \times n_{SO} \times (U_t + O_t + t_{SS} + P_M \times (P_M + 1) + 8 \times B_{op}^2 \times TC \times I_R) + \frac{1}{2} \times N_T \times n_{SO} \times (U_t + O_t + t_{SS} + 2 \times P_M + 8 \times B_{op} \times TC).$$

7 Implementation and Performance Analysis

This section is devoted to the implementation and performance study of composability checking algorithms. We use social and welfare services within the *Family and Social Services Administration* (FSSA) as an application domain for our implementation. The FSSA serves families facing issues associated with low income, mental illness, addiction, mental retardation, disability, and aging. The system, called *WebDG*, provides customized services to needy citizens.

7.1 Implementation

WebDG system is implemented across a network of *Solaris* workstations. *WebDG* architecture is organized into four layers (Fig. 8). The first layer contains a set of Oracle databases (Oracle 8.0.5) that store government and citizens' data. The second layer includes "proprietary" applications. We implemented several FSSA applications (Java JDK 1.3) including *WIC* (Women, Infant, and children), *Medicaid*, and *TOP* (Teen Outreach Pregnancy). *WIC* provides Federal grants to States for supplemental food for low-income pregnant and postpartum women. *Medicaid* is a health insurance program for specific groups of low-income people. *TOP* provides pregnant teens with childbirth and postpartum educational support.

To enable access to the aforementioned applications, we "wrapped" them into Web services. We use state-of-the-art technologies for implementing *WebDG*. Services are deployed using *Apache SOAP* (2.2). We use the WSDL language to describe *WebDG* services. WSDL descriptions are extended with semantic features defined in Section 4. We use IBM Web Service Tool Kit (WSTK) to automatically generate WSDL files for Web services from Java class files. We adopt *Systinet's WASP UDDI Standard 3.1* as our UDDI toolkit. *Cloudscape* (4.0) database is used as a UDDI registry.

The upper layer includes a *Graphical User Interface* (GUI) and *WebDG manager*. Citizens and case officers access *WebDG* via a GUI implemented using HTML/Servlet. The *WebDG manager* is at the core of the system. It is composed

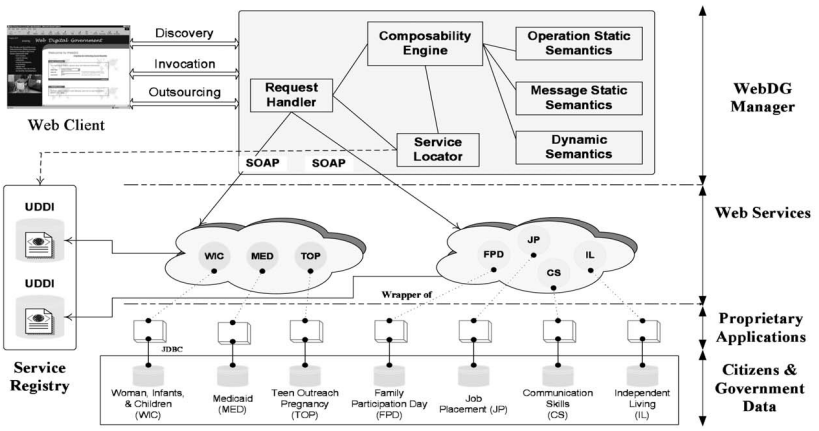


Fig. 8. WebDG architecture.

of the following main components: *Request Handler*, *Service Locator*, *Composability Engine*, *Message Semantics Controller*, *Operation Semantics Controller*, and *Dynamic Semantics Controller*. The request handler is the router of the WebDG manager. Its task depends on the type of request it receives. If the request type is "discovery," it forwards it to the service locator which implements *UDDI Inquiry Client* using WASP UDDI API. If the request type is "invocation," the request handler invokes the corresponding operation through *SOAP Binding Stub* which is implemented using Apache SOAP API. If the request type is "outsourcing," the request handler forwards the source operation (specified by the user) to the composability engine. The engine interacts with the service locator to get operations from the registry. Then, it uses the aforementioned controllers to check static semantic of messages, static semantic of operations, and dynamic semantic composability. The composability engine finally returns a set of target operations to the user.

7.2 Performance Analysis

The aim of this section is threefold. First, we show the scalability of our approach by computing the composability checking time for a large number of operations. We also compare the results of our analytical model with those obtained via experiments. Second, we assess the impact of the composability threshold τ on the number of composable operations. Third, we show the *efficacy* of our approach, i.e., the usefulness of the composability algorithms to composers.

Although we implemented the composability model in WebDG prototype, we felt it was necessary to build a simulation testbed to run the experiments. This has the advantage of allowing the generation of a large number of operations using various settings. The descriptions generated by the testbed are used to conduct our experimental study. We run our experiments on a *Sun Enterprise Ultra 10* server with a 440-MHz *UltraSPARC-IIi* processor, 1-GB of RAM, and under *Solaris* operating system. Table 4 shows the common settings for all simulation experiments. The *Confidence level* and *accuracy* are used to control the accuracy of the simulation results.[1] Users specify the values of confidence level and accuracy before starting simulation. The simulation is not complete until the expected confidence level and accuracy are achieved. Operation outsourcing requests are simulated through a random request generator. The generation of requests follows the exponential distribution. We generate 50 outsourcing requests generated ($n_{SO} = 50$) during each simulation round.

TABLE 4
Simulation Settings

N_{op}	1000 - 30000	TC	10
$Tmax$	100	*Confidence level*	0.98
P_M	50 - 100	*Confidence accuracy*	0.02
B_{op}	10 - 20		

In the first set of experiments, we evaluate the time for checking static and dynamic semantic composability (Fig. 9a). We vary the number of operations in the registry from 1,000 to 15,000 with an iteration range of 1,000. To enable a better visualization of the figure, we represent T_{SS} and T_{DS} using logarithm function. Fig. 9b depicts the composition times obtained using our analytical model (Section 6.2). Figs. 9a and 9b show that most of the composability time is spent on checking dynamic composability. Indeed, static semantic composability compares simple ontological attributes (e.g., serviceability). However, B-composability compares each business logic rules of a source operation with each business logic rules of a target operation. Each pair of business logic rules involves the comparison of two preconditions and two postconditions using the theorem prover (Fig. 7).

In the second set of experiments, we assess the impact of τ on the number of target operations (Fig. 9c). We conducted experiments for $\tau = 33\%$ and $\tau = 66\%$. The results show that the number of generated targets is higher for $\tau = 33\%$. Indeed, in this case, target operations are generated if at least 33 percent of the composability rules are satisfied. However, for $\tau = 66\%$, target operations are generated if at least 66 percent of the composability rules are satisfied.

In the third set of experiments, we measure the *efficacy* of the proposed approach (Fig. 9d). Given a set $\mathcal{O}$ of outsourcing requests, we define the *efficacy ratio* by the expression $\frac{|\mathcal{T}|}{max\_target \times |\mathcal{O}|}$, where $\mathcal{T}$ is the set of targets returned by the composition engine for all outsourcing requests, *max_target* is

1. Given N sample results $Y_1, Y_2, \ldots, Y_N$, the *confidence accuracy* is defined as *H/Y*, where H is the *confidence interval half-width* and Y is the sample mean of the results ($Y = (Y_1 + Y_2 + ... + Y_N)/N$). The *confidence level* is the probability that the absolute value of the difference between the Y and μ (the true mean of the sample results) is equal to or less than $H = t_{\alpha/2;N-1} \times \sigma/\sqrt{N}$, where $\sigma^2 = \Sigma_i (Y_i - Y)^2/(N-1)$, and t is the standard t distribution.

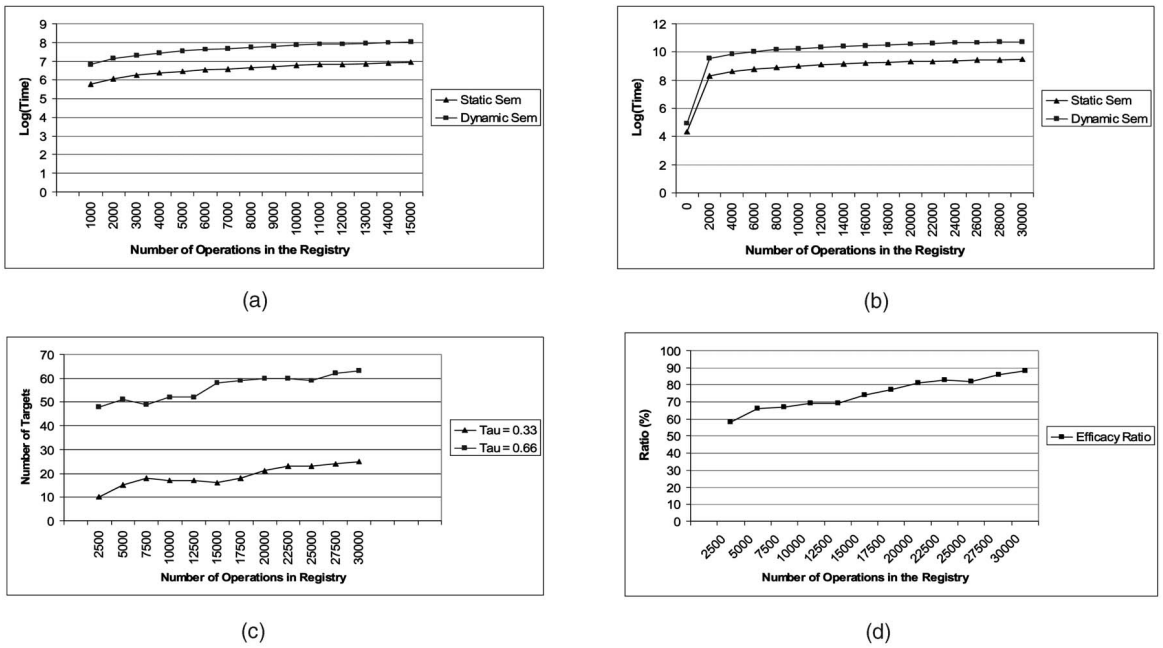


Fig. 9. Experiments and analytical model. (a) Experiments—static and dynamic semantic composition times. (b) Analytical model—static and dynamic semantic composition times. (c) Experiments—impact of threshold on composability. (d) Experiments—efficacy of the composability algorithms.

the maximum number of generated targets (Fig. 5), and $\mid \mathcal{O} \mid = n_{SO}$. This ratio gives the probability that the composition engine returns targets for a source operation. For each source operation, we consider a different value of the composability threshold τ (e.g., 0.33, 0.66). We set the variable *max_target* to 1. We also vary the number of operations available in the registry. Fig. 9d shows that the efficacy ratio varies from 58 percent to 88 percent. This means that the composition engine is able to find a composable operation for most of the outsourcing requests.

8 Related Work

In this section, we overview major techniques that are most closely related to our approach.

8.1 Composability Techniques

Several techniques have been proposed to deal with service composability. *LARKS* defines five techniques for service matchmaking: context matching, profile comparison, similarity matching, signature matching, and constraint matching [17]. These techniques mostly compare text descriptions, signatures (inputs and outputs), and logical constraints about inputs and outputs. The *ATLAS* matchmaker defines two methods to compare service capabilities described in DAML-S [15]. The first method compares functional attributes to check whether advertisements support the required type of service or deliver sufficient quality of service. The second compares the functional capabilities of Web services in terms of inputs and outputs. No evaluation study is presented to determine the effectiveness and speed of *ATLAS* matchmaker.

8.2 AI Planning Techniques for Service Composition

Another trend for dealing with the automatic service composition is the use of Artificial Intelligence planning techniques. The composition engine treats service composition as a planning problem. *SHOP2* adopts the concept of HTN (Hierarchical Task Network) as a planning methodology [19]. It decomposes tasks into smaller and smaller subtasks, until primitive tasks are found that can be performed directly. SHOP2 uses a knowledge base which contains *operators* and *methods*. Each operator is a description of what needs to be done to accomplish a primitive task. Each method describes the process of decomposing a compound task into partially ordered subtasks. *Estimated-regression* is another planning technique for service composition [11]. The situation-space search is guided by a heuristic estimator obtained by backward chaining in a relaxed problem space. The resulting space is so much smaller than situation space that a planner can build complete representation of it, called a regression graph. The regression graph reveals, for each conjunct of a goal, the minimal sequence of actions that could achieve it.

Several aspects differentiate our approach from planning-based techniques. First, our approach is rule-based while planning-based techniques are generally knowledge-based. We use a set of composability rules to determine whether Web services can be combined together. Second, we define a broader view of service composition through the notions of horizontal composition, vertical composition, and composability degree. Third, we introduce the concepts of degree and τ composability to enable partial and total composability. Finally, planning techniques generally focus on comparing pre and postconditions. Our approach adopts a more general definition of semantics (static and dynamic). It compares several service features organized into different levels.

8.3 Techniques Dealing with Service Behavior

A number of techniques dealing with service behavior have been proposed (e.g., [4], [8], [20]). These techniques deal with services whose lifecycle consists of several related calls. Models and formalisms for describing

service behaviors have been discussed, including *finite state machine*, *Mealy machine*, and *pi-calculus* [8]. Our focus in this paper was on "isolated" operations, that is, services whose lifecycle consists of independent operation invocations. The techniques described in [8] can be adopted to define dynamic semantic rules dealing with the composability of service behaviors.

A pi-calculus model for extending CORBA IDL with protocol descriptions is proposed in [4]. The model is based on the concept of role which allows the specification of the observable behavior of CORBA objects. Techniques for testing the compatibility of behaviors are also presented. *Protocol Specifications* is another approach for describing object service protocols using finite state machines [20]. This approach describes both the services offered and required by objects. It also defines techniques that allow components to be checked for protocol compatibility. In contrast to our model, the aforementioned techniques deal with objects and components *not* Web services.

9 Conclusion

We presented in this paper a framework that elicits the automated checking of service composability. We developed a multilevel composability model for semantic Web services. The model is defined by a set of rules called composability rules. Each rule specifies the constraints and requirements for checking horizontal and vertical composability. We introduced the notions of composability degree and τ-composability to cater for partial and total composability. We also proposed a set of algorithms for checking composability and implemented them in *WebDG*, a prototype for government Web services. We finally performed an analytical and experimental analysis to assess the performance of our algorithms. Future work includes the definition of behavioral rules to consider pre/postoperations during the composability process. We will also use the proposed composability model to define techniques for the automatic composition of semantic Web services.

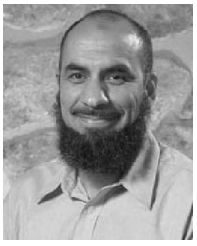
Acknowledgments

A. Bouguettaya's research was supported by the National Institutes of Health's NLM grant 1-R03-LM008140-01.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architecture, and Applications.* Springer Verlag (ISBN: 3540440089), June 2003.
[2] B. Benatallah, M. Dumas, M. Shen, and A.H. H. Ngu, "Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services," *Proc. Int'l Conf. Data Eng.*, pp. 297-308, Feb. 2002.
[3] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific Am.*, vol. 284, no. 5, pp. 34-43, May 2001.
[4] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo, "Adding Roles to CORBA Objects," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 242-260, Mar. 2003.
[5] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and Dynamic Service Composition in eFlow," *Proc. CAiSE Conf.*, pp. 13-31, June 2000.
[6] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Halevy, "Learning to Match Ontologies on the Semantic Web," *The VLDB J.*, vol. 12, no. 4, pp. 309-319, Nov. 2003.
[7] D. Fensel, *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce.* Springer Verlag (ISBN: 3540003029), Sept. 2003.
[8] R. Hull, M. Benedict, V. Christophides, and J. Su, "E-Services: A Look behind the Curtain," *Proc. Symp. Principles of Database Systems (PODS)*, pp. 1-14, June 2003.
[9] G.M. Kuper and J. Simeon, "Subsumption for XML Types," *Proc. Int'l Conf. Database Theory*, pp. 331-345, Jan. 2001.
[10] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler, "The WISE Approach to Electronic Commerce," *Int'l J. Computer Systems Science and Eng.*, vol. 15, no. 5, pp. 343-355, Sept. 2000.
[11] D.V. McDermott, "Estimating-Regression Planning for Interactions with Web Services," *Proc. Int'l Conf. Artificial Intelligence Planning Systems*, pp. 204-211, Apr. 2002.
[12] S.A. Mclraith, T.C. Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46-53, Mar. 2001.
[13] B. Medjahed, B. Benatallah, A. Bouguettaya, A. Ngu, and A. Elmagarmid, "Business-to-Business Interactions: Issues and Enabling Technologies," *The VLDB J.*, vol. 12, no. 1, pp. 59-85, May 2003.
[14] B. Medjahed, A. Bouguettaya, and A. Elmagarmid, "Composing Web Services on the Semantic Web," *The VLDB J.*, vol. 12, no. 4, pp. 333-351, Nov. 2003.
[15] T.R. Payne, M. Paolucci, and K. Sycara, "Advertising and Matching DAML-S Service Descriptions (position paper)," *Proc. Int'l Semantic Web Working Symp.*, pp. 76-78, July 2001.
[16] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, Dec. 2002.
[17] K. Sycara, M. Klush, and S. Widoff, "Dynamic Service Matchmaking among Agents in Open Information Environments," *ACM SIGMOD Record*, vol. 28, no. 1, pp. 47-53, Mar. 1999.
[18] S. Tsur, S. Abiteboul, R. Agrawal, U. Dayal, J. Klein, and G. Weikum, "Are Web Services the Next Revolution in e-Commerce? (Panel)," *Proc. Very Large Data Bases Conf.*, pp. 614-617, Sept. 2001.
[19] D. Wu, B. Parsia, J. Hendler, and D. Nau, "Automating DAML-S Web Services Composition Using SHOP2," *Proc. Int'l Semantic Web Conf.*, pp. 195-210, Oct. 2003.
[20] D.M. Yellin and R.E. Strom, "Protocol Specifications and Component Adaptors," *ACM Trans. Programming Languages and Systems*, vol. 19, no. 2, pp. 292-233, Mar. 1997.
[21] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components," *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 4, pp. 333-369, Oct. 1997.

Brahim Medjahed received the PhD degree in computer science from Virginia Tech in May 2004. He is on the Computer and Information Science Faculty at the University of Michigan at Dearborn. He received the 2004 "Outstanding Graduate Research Award" at Virginia Tech's Department of Computer Science. His research interests include data integration, semantic Web, Internet computing, Web services, and bioinformatics. He has published several papers in international journals including *IEEE Transactions on Knowledge and Data Engineering*, *The VLDB Journal*, *Distributed and Parallel Databases*, *International Journal of Cooperative Information Systems*, *IEEE Internet Computing*, *IEEE Computer*, and *IEEE Transactions on Systems, Man, and Cybernetics*. He is a member of the IEEE and the ACM.



Athman Bouguettaya is on the Computer Science faculty at Virginia Tech. He is also director of the E-Commerce and E-Government Research Lab at Virginia Tech. He is on the editorial boards of the *Distributed and Parallel Databases Journal* and the *International Journal of Cooperative Information Systems*. He guest coedited a special issue of the *IEEE Internet Computing* on database technology on the Web. He served as the program cochair of the IEEE RIDE Workshop on Web Services for E-Commerce and E-Government (RIDE-WS-ECEG'04). He served on numerous conference program committees. He is the author of more than 80 publications. His current research interests are in Web databases and Web services focusing on E-government and E-commerce. He is a senior member of the IEEE.