This report was first presented and disseminated in Workshop on Service Oriented Computing: Consequences for Engineering Requirements (SOCCER), Paris, August 2005.

Specifying Search Queries for Web Service Discovery

Shahram Esmaeilsabzali Nancy A. Day Farhad Mavaddat
Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{sesmaeil,nday,fmavaddat}@cs.uwaterloo.ca
Technical Report: CS-2011-05

Abstract

Web services are meant to be easily accessible software systems. With the emergence of Web service technologies and the presence of thousands, or perhaps millions, of Web services in the coming years, the challenge is searching for a Web service that meets a specified requirement. In this paper, we propose methods and models for effectively specifying the search queries in a Web service discovery system. We show that while search queries should be specified precisely enough to contain information for appropriate matching, they should not be too detailed. Users often do not have a clear vision of their desired Web service and a search query should be at a high enough level of abstraction. We propose two models and show how they are appropriate for capturing search queries.

1 Introduction

The ultimate goal of creating a public Web service is to be discovered and used by potential service requesters. Considering the emergence of thousands, or perhaps millions, of Web services in the coming years, it is crucial for these services to be registered and published in a standard manner. Furthermore, service requesters need to be able to search for their desired Web services efficiently.

Standards such as `UDDI` [2] aim to be a uniform way of storing repositories of Web service specifications, and to provide efficient methods for searching within them. Such search mechanisms are mainly based on the idea of decorating service specifications with metadata, and then using search APIs [9] to carry out efficient search over such metadata. This type of search can be considered a keyword-like search over different attributes of Web services, such as service types, business types, and technical finger prints.

While we believe that `UDDI` can be a useful standard for storage and categorization of Web services, we think Web service discovery should be a more involved activity than keyword-like searching. Some aspects of the behaviour of the Web services may be relevant for a search. For example, a search query may want to specify the desired Web service's inputs and outputs, and perhaps some information about message exchange order. More complex Web services can be created by combining multiple cooperating simple Web services. Existing formats, such as *OWL-S* [1], are used to create more complex Web services by using *workflow* and *dataflow* patterns to combine multiple simple Web services. It should be possible to decompose search queries as well. To ensure that the multiple parts of decomposed search query are compatible and satisfy the original query, more information needs to be described in the search query than simply values of keywords. Our goal in this work is to propose richer models for describing Web services queries and to discuss their advantages and disadvantages.

Similar to other requirement specification activities, searches for Web services can be based on *functional* or *non-functional* requirements. An example of a functional requirement is a search for an English-German dictionary Web service. If there is not an English-German dictionary, then it is possible to use an English-French and a French-German dictionary in a pipelined manner to create a composite English-German dictionary.[1] An example of a non-functional requirement is a search for an English-German dictionary Web service that is available 99% of the times. In this paper, we are interested in functional search queries.

Zaremski and Wing further categorize searching based on functional requirements into two main types: *signature/non-behavioural* [16] and *specification/behavioural* matching [17]. The former mainly investigates functions, modules, and their parameters for comparison, while the latter investigates the preconditions and effects of software artifacts and compares them against search queries. In this paper, we focus on signature/non-behavioural matching of Web services.

An important consideration for Web service discovery systems is the level of abstraction at which search queries are modelled. We believe that Web services and search

[1] This example is due to [15].

queries should not necessarily be expressed at the same level of abstraction. Service requesters often have a vague idea about the specification of the service they desire. Also, it can be argued that a precisely specified search query may ignore some potentially relevant Web services that do not satisfy the query, but are acceptable. Thus, it is crucial to be able to specify search queries at a higher level of abstraction than the Web services specifications. A helpful analogy is to imagine how hard it would be to use Web search engines, if we had to provide a precise specification of the document we require.

The form of signature matching described in [16] considers two types of matching, namely, function and module matching. Their technique focuses on the comparison between the parameters of functions, their types, and their order to find a certain software artifact. This type of matching goes beyond keyword searches, however, in terms of capturing the search queries, the queries are described with as much detail as the software artifacts' specifications in the repository.

Using ideas introduced by de Alfaro and Henzinger for interfaces [8], we characterize the level of abstraction of a model of a search query based on its amount of *statefulness*. Statefulness means how much information about the states (or behaviours) of the Web services is described in a model. If one model is less stateful than another model, then we consider it to be at a higher level of abstraction. Intuitively, a less abstract model provides a more exact specification of the states of systems than a more abstract model.

Figure 1 illustrates our proposal for how a typical Web service discovery system should work. A search query, S, written in a stateless model appropriate for search queries, may be decomposed into multiple simpler subqueries. The Web service repository provides specifications of each available service in a stateful model. A *satisfaction relation* determines whether a Web service can satisfy the query. The satisfaction relation is defined based on the semantics of the stateful (Web services) and stateless formalisms (search queries). In such a scheme, we desire the satisfaction relation to be *compositional*. A satisfaction relation is compositional, if given a query S and its constituent subqueries, $(s_1, s_2, \cdots, s_m)$, and given Web services that satisfy each subquery, *i.e.*, for all i, w_i satisfies s_i, then the composition W of the services w_i, always satisfies S. Compositional satisfaction is desired, since it allows us to divide the task of Web service discovery into multiple subtasks that can be carried out concurrently. The search query can be decomposed by the user or automatically. Also, each of the Web services themselves can be the result of composing multiple Web services. We have intentionally left the definition of composition and decomposition unspecified since these can happen through various operators.

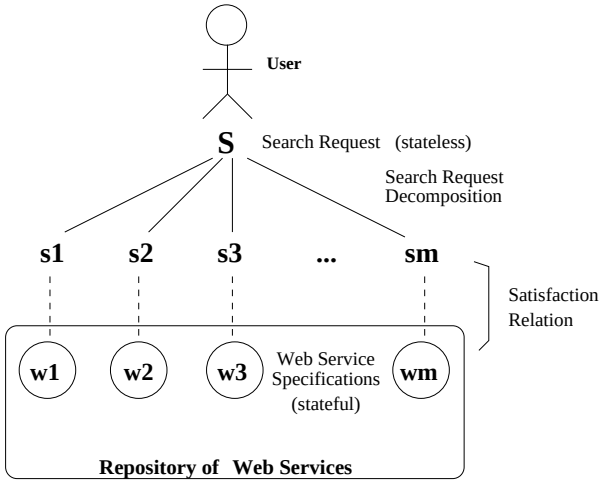

Figure 1. Web Service Discovery System

In this paper, we introduce models that are expressive enough to capture the users' functional requirements (search queries) for a composite Web service. We begin by examining the stateless model introduced by de Alfaro and Henzinger [8], called a *port dependency interface (PD)* model, and discuss its limitations for specifying Web services. The PD model is part of a class of models called *interface models* [8]; they have the properties of commutativity and associativity needed for compositionality. A PD is capable of specifying the input and output (I/O) behaviours of systems, as well as a *dependency scenario*, which is a set of dependencies among inputs and outputs. This search criteria specifies more information than possible in the models of [16].

Building on de Alfaro and Henzinger's work, we propose two new models and show how they can be used for capturing search queries in a Web service discovery system. We extend the PD model to a *multiple port dependency interface (MPD)* model. An MPD is capable of specifying multiple independent sets of dependency scenarios between inputs and outputs, as well as outputs and inputs. Next, we extend MPD to an *ordered port dependencies (OPD)* model, which is similar to MPD except that dependency pairs are placed in a sequence allowing a more precise search specification. OPD can provide a more succinct way to describe I/O dependencies compared to MPD, however, it lacks the associativity property in composition, necessary to be an interface model.

Since we are concentrating on modelling search queries, we do not impose any particular requirements on the format or technology of the repository in Figure 1; it suffices for us that the specifications are stored in a stateful manner. We briefly discuss the suitability of *interface automata* [7], a stateful interface model, as a candidate for specifying

the behaviours of Web services. Other approaches have also been proposed for modelling Web service functionality, *e.g.*, Petri Nets [12], Finite State Machines [4], and Statecharts [5].

The main contributions of our work are as follows: (1) We separate the models used for search queries from those used for modelling Web services; (2) We introduce the idea of using stateless models with I/O dependencies for describing Web service search queries to raise the level of abstraction of these descriptions; and (3) We introduce and discuss two new stateless models for modelling Web services search queries and discuss their advantages and disadvantages.

2 Background

The class of models called *interface models* [8], introduced by de Alfaro and Henzinger, contains concise models meant to specify *how* systems can be used. The two main characteristics of interface models are that they assume *helpful environments* and support *top-down* design. A *helpful environment* provides proper inputs for an interface and receives all of its outputs. As such, interfaces are optimistic, and do not usually specify all possible behaviours of the systems. For example, they often do not include fault scenarios. *Top-down design* is based on a notion of refinement, which relates two instances of a model. A refinement of a model can be substituted for the original.

These characteristics make interface models suitable for describing Web services. Web services rely on a helpful environment in the form of a coordination mechanism that appropriately invokes their functionality, provides their required inputs, and receives all of their possible outputs. In the context of Web services, top-down design means that once a Web service is specified as the composition of multiple interface models, if we replace one of the interfaces with a more refined version of it, the resulting composition also refines the initial model. For search queries of Web services, refinement is less useful. The composition of multiple search queries should equal the original query rather than just refine it.

In [8], interface models are categorized into *stateless* and *stateful* models. Stateless models mainly specify the input and output behaviours of systems, while stateful systems specify the internal behaviours of systems by modelling the temporal order of events. We believe that stateless interface models are appropriate models for capturing search queries because they provide a richer model than keyword search, but do not require all the details needed in a specification of a Web service.

Formally, a model is a well-formed interface model if the binary operations *composition* and *refinement* are defined. Composition must be a commutative and associative operation. To support top-down design, for three interfaces F, F', G, and the composition of F and G, $F \parallel G$, if F' refines F, *i.e.*, $F' \preceq F$, then:

$$(F' \parallel G) \preceq (F \parallel G)$$

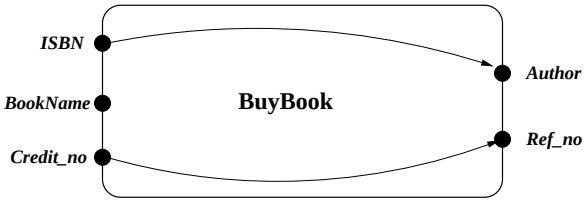

Figure 2. PD BuyBook

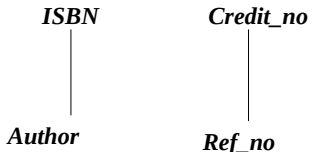

Figure 3. Partial order on ports of dependency scenario of PD of Figure 2

de Alfaro and Henzinger introduce the *port dependency interface* (PD) model, a stateless interface model. PD is the first model we considered using for Web service search query specification. Figure 2 is a graphical representation of a PD, *BuyBook*, used to describe a Web service search query for a book buying Web service. [2] The named, filled circles represent the *ports*, which are the inputs and outputs of *BuyBook*. The ports on the left are input ports, and ports on the right are output ports. *BuyBook* represents a search query for a Web service that can carry out a book purchase by receiving an ISBN and a credit card number; the output of the Web service is the author of the book as well as the reference number for the successful credit card transaction. The arrows between the ports represent the dependencies between input and output ports. A dependency pair states that the value on a certain input port influences the value of a certain output port.

Dependency pairs in a PD establish an irreflexive partial order. Figure 3 shows the simple partial order that is implied by the dependency pairs in Figure 2. A partial order on ports can express concurrency [14]. Any set of ports in a set of dependency pairs that are not related by the partial order can be used concurrently. Thus, a set of dependency pairs is a powerful tool to express allowed concurrencies in a search query.

Notice that no dependencies are shown for the *BookName* port. Ideally, we would have liked to state that the

[2] The graphical representation, as shown in Figure 2, is not a part of the PD model.

service can produce the outputs from both an *ISBN* and *Credit_no*, or a *BookName* and *Credit_no*. However, PD is not expressive enough to capture the "or" semantics. In the next section we will introduce our MPD model, which has this capability.

Formally, the definition of a PD [3] is:

Definition 2.1 *A port dependency interface model* $F = \langle I_F, O_F, \kappa_F \rangle$*, consists of:*

- I_F*: The set of input ports.*
- O_F*: The set of output ports, which is disjoint from the input ports. We denote* $P_F = I_F \cup O_F$ *as the set of all ports.*
- $\kappa_F \subseteq I_F \times O_F$*: The I/O dependency relation of the interface, which we call a dependency scenario.*

■

The composition of two PDs F and G, $F \parallel G$, is defined if they are *composable*, meaning they do not use any of the same inputs or outputs, *i.e.*, $P_F \cap P_G = \emptyset$. Composition of two composable PDs is the union of their elements:

$$F \parallel G = \langle (I_F \cup I_G), (O_F \cup O_G), (\kappa_F \cup \kappa_G) \rangle$$

Renaming of ports in stateless interface models, including PD, is done using a *connection function* that maps an interface and a renaming function to a new interface. In this paper, for the sake of simplicity, we present our models without a description of connection functions; instead, we use shared names of ports as an implicit way to model connections. The same models are presented with explicit connection functions in [10].

3 Multiple Port Dependency Model

In this section, we introduce our *multiple port dependency interface* (MPD) model and define composition on MPD. MPD, similar to PD, is a stateless interface model. Compared to a PD, in an MPD we can specify multiple dependency scenarios, *i.e.*, multiple sets of I/O dependencies. Different I/O dependency scenarios can be used to specify search queries for Web services with multiple alternative behaviours. We believe MPD is an appropriate formalism for specifying such search queries; while it is simple to use, it is expressive enough to capture I/O dependencies as well as concurrent behaviours of systems.

Additionally, within a dependency scenario, MPD allows output-input dependency pairs rather than just the input-output pairs of PD. Output-input dependencies can be useful when we want to model queries for Web services with the `Out-In` message passing pattern in WSDL (Web Services Description Language) 2.0 [6]. `Out-In` represents the situation where a Web service sends a message to the service requester and the service requester in turn sends back a message.

[3] The definition of a PD also includes a set of available ports, which are necessary for refinement. Since we are not interested in refinement for search queries, we omit this part of the definition in all the models presented in this paper. This model element plus definitions of refinement for all models presented in this paper can be found in [10].

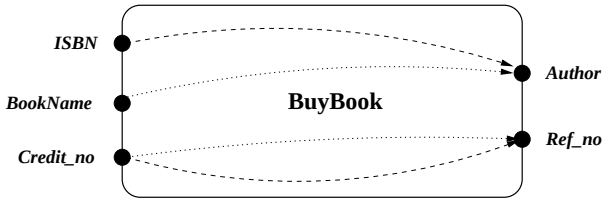

Figure 4. MPD BuyBook

Dependency pairs of each dependency scenario establish an irreflexive partial order and thus can express concurrency. Since MPDs can have multiple dependency scenarios, it is possible to express different concurrency scenarios in a search query.

BuyBook in Figure 4 is an MPD with two dependency scenarios that are represented by dashed and dotted lines. Each dependency scenario represents one of the possible ways that the desired Web service should work. The "dashed" scenario receives the ISBN of a book and a credit card number, and returns the author of the book and a reference number for the credit card number transaction. The "dotted" scenario is similar but receives a book name instead of its ISBN. The Web service satisfying this search query must be capable of satisfying both dependency scenarios.

To overcome some of the restrictions on PD composition, in MPDs, we introduce the concept of *shared ports*. The set of shared ports between two MPDs are the ports with the same names. While in PD, two interfaces with shared ports are not composable, in MPD if each of the ports in the set of shared ports is an output of one MPD and input of another, they can be composed. Intuitively, shared ports allow two MPDs to communicate through ports with the same names.

As an example, consider MPDs F and G in Figure 5; each MPD has one dependency scenario shown by solid lines. They are composable MPDs since their shared ports, b and d, are both the input of one and the output of the other. If they do not have any shared ports, composition of two MPDs is the union of their elements. In the presence of shared ports, the shared ports and dependency pairs defined on them are removed from the composition; new dependency pairs are introduced using transitivity between dependency pairs on shared ports. In Figure 5, $F \parallel G$ is the composition of F and G; the (a, f) dependency is a new

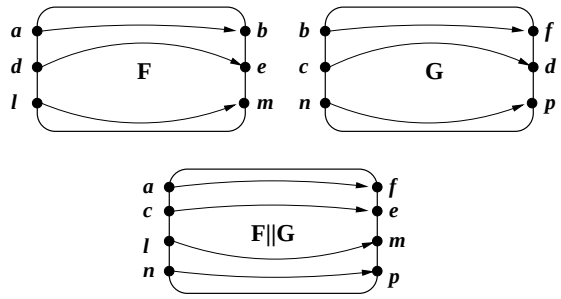

Figure 5. MPD Composition

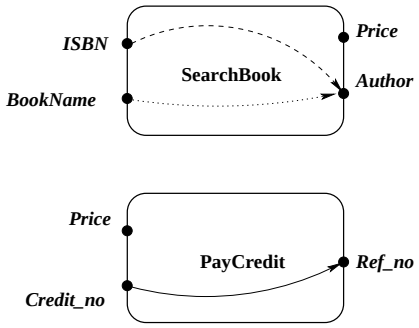

Figure 6. Decomposition of MPD BuyBook

dependency pair created from (a, b) of F and (b, f) of G.

The shared ports of two composable MPDs are removed and stored in the set of *hidden ports* of their composition. The hidden ports state that a certain port is an internal port of the system. We require that two composable MPDs not have any common internal ports.

We can decompose an MPD into two MPDs. As an example, the two MPDs in Figure 6 are the decomposition of MPD *BuyBook* in Figure 4. If there is no Web service that can satisfy *BuyBook*, we may be able to find Web services that satisfy *SearchBook* and *PayCredit* independently, and then compose those Web services to satisfy *BuyBook*, assuming that our satisfaction relation is compositional. The port *Price* becomes a hidden port in the composition.

In the following subsections, we formally define MPD and then describe its composition operation. The reader can refer to [10] for our proofs that MPD is a well-formed interface model.

3.1 Formal Description of MPD

Formally, the definition of an MPD is:

Definition 3.1 *A multiple port dependency interface* $F = \langle I_F, O_F, H_F, \mathcal{U}_F \rangle$ *consists of:*

- I_F*: The set of input ports.*
- O_F*: The set of output ports, which is disjoint from the input ports. We denote* $P_F = I_F \cup O_F$.
- H_F*: The set of hidden ports, which must be disjoint from* $I_F \cup O_F$.
- $\mathcal{U}_F$*: The set of I/O dependency scenarios. Each dependency scenario* $u_F \in \mathcal{U}_F$ *is a set of dependency pairs, and for each dependency pair* $(a, b) \in u_F$*, the following conditions hold:*
  - $(a, b) \subseteq ((I_F \times O_F) \cup (O_F \times I_F))$ *(It relates inputs to outputs or outputs to inputs.)*
  - $\forall\, c \in P_F \Rightarrow \nexists (b, c) \in u_F$ *(A port cannot both be influenced by a port and influence another port. This disallows circular dependencies.)*
  - $\nexists d \in P_F \cdot (d \neq a) \wedge (d, b) \in u_F$ *(A port is not influenced by more than one port.)*

■

3.2 MPD Composition

MPD composition combines the ports and dependency scenarios of two composable MPDs, and stores the shared ports that become hidden in the set of hidden ports of the resulting MPD. The *shared ports* between two MPDs, F and G, are their common input and output ports, *i.e.*, $SharedPorts(F, G) = P_F \cap P_G$. Two MPDs may be composed if each of their shared ports is an input of one and an output of another. This restriction avoids the ambiguity created by having different input and output ports of the same name. We also require that two composable MPDs not have any hidden ports in common since hidden ports are the result of having had the same input and output ports in two MPDs that were composed to create this MPD. Composing two MPDs with the same hidden ports results in an ambiguity as to the source of the hidden ports. Similar composability criterion has been proposed in interface automata [7] and I/O automata [11]. Formally:

Definition 3.2 *Two MPDs,* $F = \langle I_F, O_F, H_F, \mathcal{U}_F \rangle$ *and* $G = \langle I_G, O_G, H_G, \mathcal{U}_G \rangle$ *are composable, if and only if the following three conditions hold:*

1. $\forall\, a \in SharedPorts(F, G) \cdot$
   $((a \in I_F) \wedge (a \in O_G)) \vee ((a \in I_G) \wedge (a \in O_F))$
   (A shared port is the input of F *and the output of* G *or vice versa.)*
2. $H_F \cap H_G = \emptyset$ *(*F *and* G *do not have hidden ports in common.)*

3. *For each $u_F \in \mathcal{U}_F$, $u_G \in \mathcal{U}_G$, and $L = u_F \cup u_G$, the transitive closure of L, L^*, satisfies:*

$$\forall (x, y) \in L^* \Rightarrow (y, x) \notin L^*$$

(No combination of dependency scenarios of F and G results in circular dependencies.)

■

Next, we define composition for MPDs:

Definition 3.3 *The composition of two composable MPDs, F and G, $F \parallel G$, is an MPD and is defined as:*

- $I_{F\|G} = (I_F \cup I_G) \backslash SharedPorts(F, G)$
- $O_{F\|G} = (O_F \cup O_G) \backslash SharedPorts(F, G)$
  We denote $P_{F\|G} = I_{F\|G} \cup O_{F\|G}$.
- $H_{F\|G} = H_F \cup H_G \cup SharedPorts(F, G)$
- *For each $u_F \in U_F$ and $u_G \in U_G$, create a $u_{F\|G} \in \mathcal{U}_{F\|G}$, such that:*

$$u_{F\|G} = (u_F \cup u_G)^* \backslash ((SharedPorts(F, G) \times P_{F\|G}) \cup (P_{F\|G} \times SharedPorts(F, G)))$$

Where "\" means set difference.

(Each dependency scenario is the result of combining dependency pairs of two dependency scenarios belonging to F and G with dependency pairs on shared ports being removed.)

■

4 Ordered Port Dependency Model

While the MPD model is an expressive tool for capturing search queries, it has its limitations. It is not possible to specify a total order on different ports of an MPD. As an example, consider the $\{(ISBN, Author), (Credit\_no, Ref\_no)\}$ dependency scenario in the MPD *BuyBook* of Figure 4. It is not clear whether $(ISBN, Author)$ or $(Credit\_no, Ref\_no)$ should happen first, or whether they are concurrent.

To impose an order on the ports, we could either have dependency pairs between inputs or between outputs, *e.g.,* $(ISBN, Credit\_no)$, or we could define an order between dependency pairs, *e.g.,* $\langle (ISBN, Author), (Credit\_no, Ref\_no) \rangle$. In our *ordered port dependency* (OPD) model [4], we choose the second approach. OPD is similar to MPD except that instead of having dependency scenarios, which are sets of dependency pairs, it has *dependency rules*, which are sequences of dependency pairs. A dependency rule is a total order on the pairs of dependencies.

[4] OPDs are called enhanced port dependency (EPD) models in [10].

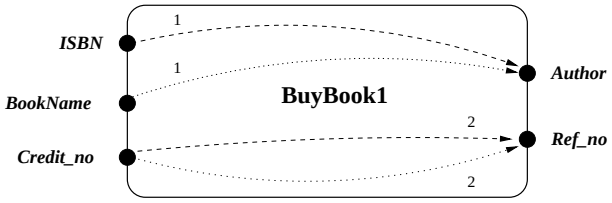

Figure 7. OPD BuyBook1

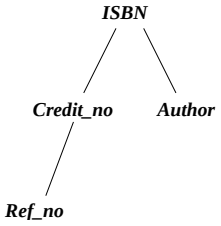

Figure 8. Partial order on ports of dashed dependency rule of Figure 7

Figure 7 shows the OPD *BuyBook1*. It is very similar to the search query of MPD *BuyBook* in Figure 4. Two dependency rules are represented by the dashed and dotted lines. The numbers on dependency pairs represent the order of dependency pairs in the dependency rule. Figure 8 represents the partial order resulting from the "dashed" dependency rule of OPD *BuyBook1* in Figure 7. The order implies that *ISBN* happens before *Author*, *Credit_no* appears before *Ref_no*, and *ISBN* happens before *Credit_no*; the first element of a pair comes before the first element of the next pair in the partial order. The partial order of ports of an OPD always has a single root.

To make it possible to specify a total sequential order on ports, OPDs can have *identity dependencies*, *i.e.,* a dependency between a port and itself. An identity dependency forces a port to be used at a particular point in the dependency rule. For example, OPD *BuyBook2*, in Figure 9, uses identity dependencies to specify a very similar search query as OPD *BuyBook1* in Figure 9, except that a total order is imposed on the ports. Figure 10 shows the total order of ports given by the dashed dependency rule of OPD *BuyBook2*. Also, OPD *BuyBook2* provides an easier decomposition scheme than OPD *BuyBook1*; we will see more about composition and decomposition later in this section.

For search queries, identity dependencies can be used to capture `In-Only` and `Out-Only` message passing pat-

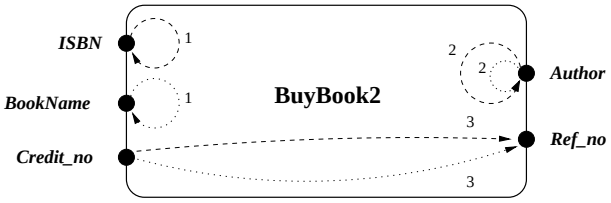

Figure 9. OPD BuyBook2

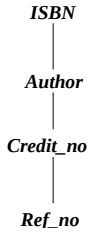

Figure 10. Total order on ports of dashed dependency rule in Figure 9

terns in WSDL 2.0 [6]. These patterns model the situations where a Web service requester (`In-Only`) or provider (`Out-Only`) sends a message without receiving any.

Having an order on the pairs of dependencies decreases the concurrency expressiveness of OPD, in comparison with MPD, because OPD imposes more order on dependency pairs and makes their concurrency less likely. On the other hand, OPDs provide a way of expressing sequentiality lacking in MPDs. We do not entirely loose the ability to express concurrency in OPD. For example, in the partial order in Figure 8 the order that *Ref_no* and *Author* are produced is not specified and these operations can be carried out concurrently.

Next, we give a formal description of OPD and describe the composition of two OPDs.

4.1 Formal Description of OPD

Formally, the definition of OPD is as follows:

Definition 4.1 *An ordered port dependency model* $F = \langle I_F, O_F, H_F, \mathcal{V}_F \rangle$ *consists of:*

- I_F*: The set of input ports.*
- O_F*: The set of output ports, which is disjoint from the set of input ports. We denote* $P_F = I_F \cup O_F$.
- H_F*: The set of hidden ports, disjoint from* $I_F \cup O_F$.
- $\mathcal{V}_F$*: The set of* dependency rules. *Each* $v_F \in \mathcal{V}_F$ *is a sequence of distinct dependency pairs. For each dependency pair* $(a, b) \in v_F$ *one of the following two sets of conditions must hold:*

1. - $(a, b) \subseteq ((I_F \times O_F) \cup (O_F \times I_F))$ *(It relates inputs to outputs or outputs to inputs.)*
   - $\forall\, c \in P_F \Rightarrow \nexists (b, c) \in v_F$ *(A port cannot both be influenced by a port and influence another port. This disallows circular dependencies.)*
   - $\nexists d \in P_F \cdot (d \neq a) \wedge (d, b) \in v_f$ *(A port is not influenced by more than one port.)*
2. - $(a = b) \wedge (a \in P_F)$ *(It is an identity dependency, i.e., it relates one port to itself.)*
   - $(\nexists (a, c) \in v_F \cdot c \neq a) \wedge (\nexists (c, a) \in v_F \cdot c \neq a)$ *(A port that has an identity dependency cannot take part in any other dependency pair.)*

■

4.2 OPD Composition

The composition of two OPDs combines the ports and dependencies of the two OPDs. The composition operator creates a new dependency rule by combining a dependency rule from each of the two OPDs being composed. It has to somehow mix the dependency pairs such that synchronization between shared ports occurs. Rather than considering all interleavings, we believe it is sufficient to consider only linear orders in which one or the other model runs until synchronization between shared ports occurs. Including all interleavings makes the model unusable because of its size, plus requires the Web service to satisfy all interleavings.

The criteria for composability for OPDs is exactly the same as that for MPDs (Definition 3.2), except that in the condition to avoid circularity, we consider each dependency rule as a set rather than a sequence.

The dependency rules of a composed MPD are created by composing all combinations consisting of a dependency rule from each of the two models being composed. To ensure commutativity, we compose the two rules in either order and include both resulting dependency rules. To compose dependency rule A with dependency rule B, we walk along the dependency pairs of A sequentially, and include these pairs in the resulting dependency rule until we reach a point where the second port in a dependency pair of A can synchronize with the first port of a dependency pair in B, *i.e.,* there is a shared port. At this point, we include any dependency pairs in B prior to the shared port. Next, we create a new dependency pair through transitivity of the two dependency pairs on the shared port. Then, we return

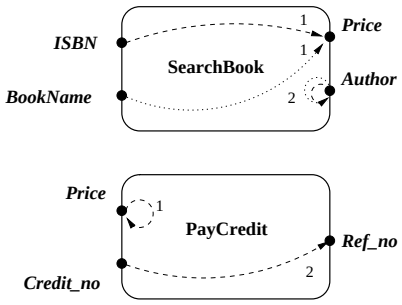

Figure 11. Decomposition of OPD BuyBook2

to A's dependency rule and repeat these operations until we reach the end of both sequences. This method ensures that at most two dependency rules result from the composition of two dependency rules. Some dependency rules are not composable. The mathematical definition of this composition can be found in [10]

OPD composition is restrictive in comparison with MPD. In MPD, dependency scenarios have the chance to synchronize in any possible order; in OPD, on the other hand, the synchronization can only happen while we traverse forward along the two dependency rules. As an example consider the two dependency rules, $r = \langle(a,b),(c,d),(e,f),(g,h)\rangle$ and $s = \langle(l,m),(b,e),(n,o),(f,p)\rangle$; (a,p) is created through the $\langle(a,b),(b,e),(e,f),(f,p)\rangle$ chain of synchronization. By traversing r first, the resulting dependency rule is $\langle(l,m),(c,d),(n,o),(a,p),(g,h)\rangle$. If we start traversing s first, there is no way that we can synchronize on all shared ports.

As an example of OPD composition, consider the two OPDs in Figure 11. They are the decomposition of OPD *BuyBook2* in Figure 9. The dashed dependency rule of OPD *BuyBook2*, $\langle(ISBN,ISBN),(Author,Author),$ $(Credit\_no,Ref\_no)\rangle$, is created by combining the dashed dependency rule of OPD *SearchBook* with the dependency rule of OPD *PayCredit*. Dependency pair $(ISBN,ISBN)$ is created by the transitivity of $(ISBN,Price)$ belonging to OPD *SearchBook*, with $(Price,Price)$ belonging to OPD *PayCredit*. This transitive sequence of dependency pairs is a special case because it ends with an "identity" dependency pair; by definition the result of such a chain of synchronizations is an identity dependency on the first port of the first dependency pair. Other dependency pairs that are not defined on shared ports, *i.e.,* $(Author,Author)$ and $(Credit\_no,Ref\_no)$, are inserted into the sequence appropriately. Similarly, the second dependency rule of OPD *BuyBook2* can be computed. Because identity dependency pairs affect only one port, it is often the case that their presence makes it easier to find a decomposition scheme.

OPD composition is not associative, and thus OPD is not a well-formed interface model. Non-associativity in composition arises since we do not allow the interleaving of all possible behaviours in our composition operation; as such our composition combines dependency rules in such a way that the order of combinations affects the resulting dependency rule. We believe that when expressiveness is the major concern, and not composition, then the OPD model is appropriate for capturing search queries.

5 Workflow Patterns

A composite Web service is a set of simple Web services that are connected to each other through workflow patterns. Both MPD and OPD allow the flow of data between different interfaces via a shared port mechanism, *i.e.,* they offer a mechanism to describe data flow. The composition operation for MPD and OPD can be considered roughly equivalent to a "parallel" workflow pattern. In this section, we consider whether it is possible to simulate the workflow patterns of choice and sequence with MPDs and/or OPDs. Choice and sequence are among the basic workflow patterns that are often used in specifying systems [3].

5.1 Sequence

For OPDs, a sequence operation is represented using a concatenation operator, which describes the sequential execution of two OPDs. Because sequence does not support communication between the involved models, it can only be defined when there are no shared ports between the two models.

Definition 5.1 *The* concatenation *of two OPDs,* $F = \langle I_F, O_F, H_F, \mathcal{V}_F\rangle$ *and* $G = \langle I_G, O_G, H_G, \mathcal{V}_G\rangle$*, is an OPD,* $FG = \langle I_{FG}, O_{FG}, H_{FG}, \mathcal{V}_{FG}\rangle$*, if* $SharedPorts(F,G) = \emptyset$ *and*

- $I_{FG} = I_F \cup I_G$ *(*FG *receives both* F*'s and* G*'s inputs.)*
- $O_{FG} = O_F \cup O_G$ *(*FG *generates both* F*'s and* G*'s outputs.)*
- $H_{FG} = H_F \cup H_G$ *(The hidden ports of* FG *is the union of the hidden ports of* F *and* G*.)*
- *For each* $v_F \in \mathcal{V}_F$ *and* $v_G \in \mathcal{V}_G$*, create a* $v_{(FG)} \in \mathcal{V}_{FG}$ *such that* $v_{(FG)} = v_F.v_G$ *(All dependency rules of* FG *consist of a dependency rule of* F *immediately followed by one of* G*'s dependency rules.)*

■

The sequence workflow pattern for MPD is not relevant since, as opposed to OPD, it lacks the notion of order in its dependency pairs.

5.2 Choice

The *choice* of two OPDs F and G, $F \curlywedge G$, means that at any point of time either F or G executes. Unlike sequence, we allow the set of shared ports of two OPDs to have common ports on their input or output ports since at each point in time we only deal with either F or G and thus common names cause neither ambiguity nor circular dependencies.

Definition 5.2 *The* choice *of two OPDs,* $F = \langle I_F, O_F, H_F, \mathcal{V}_F \rangle$ *and* $G = \langle I_G, O_G, H_G, \mathcal{V}_G \rangle$*, is an OPD,* $F \curlywedge G = \langle I_{F \curlywedge G}, O_{F \curlywedge G}, H_{F \curlywedge G}, \mathcal{V}_{F \curlywedge G} \rangle$*, if for all* $x \in SharedPorts(F, G)$*, either* $(x \in I_F) \wedge (x \in I_G)$ *or* $(x \in O_F) \wedge (x \in O_G)$*, and*

- $I_{F \curlywedge G} = I_F \cup I_G$ *($F \curlywedge G$ receives both F's and G's inputs.)*
- $O_{F \curlywedge G} = O_F \cup O_G$ *($F \curlywedge G$ generates both F's and G's outputs.)*
- $H_{F \curlywedge G} = H_F \cup H_G$ *(The hidden ports of $F \curlywedge G$ include the hidden ports of both OPDs.)*
- $\mathcal{V}_{F \curlywedge G} = \mathcal{V}_F \cup \mathcal{V}_G$*. (Each of $F \curlywedge G$'s dependency rules is one of F's or G's dependency rules.)*

■

The choice operation for MPD can be defined in a similar way.

6 Summary and Future Work

We have introduced a method, along with two models, for specifying search queries for a Web service discovery system. We described why search queries should not necessarily be specified at the same level of abstraction as Web services themselves. We introduced two models that are capable of specifying search queries at a high level of abstraction. The first model, the multiple port dependency interface (MPD) model, is a stateless model with sets of dependencies between inputs and outputs. The second one, the ordered port dependencies (OPD) model, provides the means to specify more precisely the order of the inputs and outputs. We introduced composition for our models, which allows them to be used for specifying composite search queries by composing simple search queries.

The next step is to assess how our search query models work with a repository of Web service specifications and a satisfaction relation. One possible formalism for specifying Web services is interface automata (IA) [7]. For example, a specification of a Web service for buying books is given in Figure 12 as an IA, which is an automaton machine that

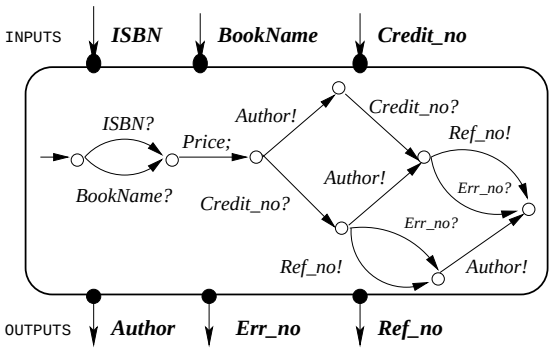

Figure 12. Interface Automata BuyBook

changes its states by input(?), output(!), or internal(;) actions. This Web service satisfies the requirements of the MPD in Figure 4, and of the OPDs in Figure 7 and Figure 9. It provides the set of input and output actions required by the search queries and satisfies the required dependencies. One can see the difference in level of abstraction between the stateless models of PD, MPD, and OPD, and stateful model of IA in this example.

In this paper, the names of ports are strings that are compared to other port names for equality, however, a richer meaning could be associated with these names. The satisfaction relation could relate port names semantically to do a more meaningful comparison between names. Existing semantic matching engines, such as the one proposed in [13], can be used for this purpose.

Finally, we believe that the graphical representations of MPD and OPD can be used to create a simple and intuitive GUI to help users create search queries in a Web service discovery system. In [10], we give a version of our models with explicit connection functions, which provides an easy way to direct the dataflows of the system graphically between different interfaces. A user can easily draw boxes and ports, and using composition and workflow operators, and connection functions relate different boxes to create queries for Web services. We plan to use the GUI as a platform for experimenting with user-guided decomposition of search queries.

References

[1] OWL-S: Semantic markup for Web services. W3C Member Submission, Nov 2004.

[2] UDDI version 3.0.2 UDDI Specification Technical Committee draft. Oct. 2004.

[3] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed Parallel Databases*, 14(1):5–51, 2003.

[4] Boualem Benatallah, Marlon Dumas, Quan Z. Sheng, and Anne H. H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 297–308. IEEE Computer Society, 2002.

[5] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proc. of the 1st Int. Conf. on Service Oriented Computing (ICSOC 2003)*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.

[6] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer, and Sanjiva Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. W3C Working Draft, May 2005.

[7] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In Volker Gruhn, editor, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)*, volume 26, 5 of *Software Engineering Notes*, pages 109–120. ACM Press, 2001.

[8] Luca de Alfaro and Thomas A. Henzinger. Interface Theories for Component-Based Design. In *Proceedings of the First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer-Verlag, 2001.

[9] UDDI Open Draft. UDDI Version 2.04 API Specification. OASIS Standard, 2002.

[10] Shahram Esmaeilsabzali. An Interface Approach to Discovery and Composition of Web Services. Master of Mathematics, School of Computer Science, University of Waterloo, 2004.

[11] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 519–543, 1987.

[12] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International Conference on World Wide Web*, pages 77–88, 2002.

[13] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333–347. Springer-Verlag, 2002.

[14] Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[15] Evren Sirin, James Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web Services: Modeling, Architecture and Infrastructure Workshop in ICEIS 2003*, April 2003.

[16] Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1996.

[17] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.