Analysis and Management of Web Service Protocols

Boualem Benatallah[1], Fabio Casati[2], and Farouk Toumani[3]

[1] CSE, UNSW, Sydney NSW 2052, Australia
boualem@cse.unsw.edu.au
[2] Hewlett-Packard Laboratories, Palo Alto, CA, 94304 USA
casati@hpl.hp.com
[3] LIMOS Laboratory, ISIMA, UBP Clermont-Ferrand, France
ftoumani@isima.fr

Abstract. In the area of Web services and service-oriented architectures, business protocols are rapidly gaining importance and mindshare as a necessary part of Web service descriptions. Their immediate benefit is that they provide developers with information on how to write clients that can correctly interact with a given service or with a set of services. In addition, once protocols become an accepted practice and service descriptions become endowed with protocol information, the middleware can be significantly extended to better support service development, binding, and execution in a number of ways, considerably simplifying the whole service life-cycle. This paper discusses the different ways in which the middleware can leverage protocol descriptions, and focuses in particular on the notions of protocol compatibility, equivalence, and replace-ability. They characterise whether two services can interact based on their protocol definition, whether a service can replace another in general or when interacting with specific clients, and which are the set of possible interactions among two services.

1 Introduction

Web services, and more in general service-oriented architectures (SOAs), are emerging as the technologies and architectures of choice for implementing distributed systems and performing application integration within and across companies' boundaries. The basic principles of SOAs consist in modularizing functions and exposing them as services, that are typically specified using (de jure or de facto) standard languages and interoperate through standard protocols. Web service technology is characterized by two trends that were not part of conventional (e.g., CORBA-like) middleware services and that are relevant to the topics discussed in this paper. The first is that, from a technology perspective, all interacting entities are considered to be (Web) services, even when they are in fact requesting and not providing services. This allows uniformity in the specification language (for example, the interface of both requestor and providers will be described using the Web Services Description Language – WSDL) and uniformity in the development and runtime support tools.

P. Atzeni et al. (Eds.): ER 2004, LNCS 3288, pp. 524–541, 2004.

The second trend, that is gathering momentum and mindshare, is that of including, as part of the service description, not only the service interface but also the *business protocol* supported by the service, i.e., the specification of which message exchange sequences (called *conversations* in the following) are supported by the service [3]. This is important, as it rarely happens that service operations can be invoked at will independently from one another. The interactions between clients and services are always structured in terms of a set of operation invocations, whose order typically has to obey certain constraints for clients to be able to obtain the service they need. In the following, we use the term *external specification* to refer to the combination of the interface and business protocol specifications, that define the externally visible behavior of a service [1, 12]. In addition to the business protocol[1], a service may be characterized by other protocols, such as security (e.g., trust negotiation) or transaction protocols that also need to be exposed as part of the service description so that clients know how to interact with a service [3, 10].

If two or more services need to interoperate, their protocols must be *compatible*. For example, a bookseller's business protocol may require customer's Web services to first invoke the *orderBook* operation and then the *makePayment* operation. If a requestor wishes to interact with this service, then its business protocol will need to include the invocation of the *orderBook* operation followed at some point by the invocation of the *makePayment* operation. If this is not the case, then the interaction between the two entities will result in an error. Hence, it is essential that requestors are only bound, statically or dynamically, to providers that have compatible protocols.

This paper analyzes protocol compatibility and similarity in Web services. In particular, we define and characterize different types of protocol compatibility, corresponding to different capabilities of services to interoperate, and we show how, given two services and their external specifications, it is possible to formally identify their compatibility level. In addition, we discuss similarities and differences between protocols, to understand if two services exhibit the same behavior or if one can be used instead of another when serving a certain client. In doing the analysis, our motivation and goal is to devise protocol management primitives that support and simplify service development. This complements our earlier efforts aiming at designing and developing a complete CASE tool supporting the Web service lifecycle [3, 2, 10]. Indeed, and as discussed in this paper, the primitives presented here can be used by service development and runtime environment to: i) assist developers in creating and evolving Web services that are compatible with other services of interest or with standard protocol specifications; ii) identify (statically or dynamically) services that can interoperate with a given service; iii) manage non-compatibility situations.

This paper does not discuss other aspects that are in general relevant to identifying whether two services can interact to achieve the desired goals. For example, we do not deal with quality of service issues, or with structural and semantic interoperability of messages [6]. While we believe that these issues are also important, the (syntactic) protocols compatibility and similarity analysis

[1] In this paper we will use "business protocol" and "protocol" interchangeably.

discussed here is complex enough in itself to deserve the whole paper (and indeed many aspects still remain to be addressed). Finally, we observe that, although to make the presentation more concrete we will introduce the concepts based on a specific protocol language, the results presented here can be applied to any protocol language, such as WSCI or BPEL.

The outline of the paper is as follows: Section 2 introduces a protocol model and some notations and concepts used throughout the paper. Section 3 defines a collection of protocol management operators that allow understanding commonalities and differences between protocols, as well as whether two protocols can interact with each other. Section 4 introduces compatibility and similarity classes, and shows how the model and operators developed in the previous sections can be used to analyze and understand the kind of compatibility or similarity that two protocols exhibit. Finally, Section 5 concludes the paper with a discussion of possible applications of the proposed protocol analysis.

2 Preliminaries

2.1 Business Protocols Modeling

Following our previous work [3], we choose to model a service business protocol (protocol for short) as a non-deterministic finite state machine, where the states represent the different phases that a service may go through during its interaction with a requestor. Transitions are triggered by messages sent by the requestor to the provider or vice versa (hence, transitions are labeled with either input or output messages). A message corresponds to the invocation of a service operation or to its reply. Note that each service may be simultaneously involved in several message exchanges (conversations) with different clients, and therefore can be characterized by multiple concurrent instantiations of the protocol state machine. The purpose of the protocol is essentially to specify the set of conversations that are supported by the service. The reason for using a state machine-based model is because it a formalism that is fairly easy to understand for users, it is suitable to describe reactive behaviors, and it has the notion of state which is useful for monitoring service executions. Furthermore, there are a number of models and tools (some developed by the authors [2]), that enable protocol modelling by means of state machines. The need for non-determinism comes from the observation that a service may respond in different ways to a certain message, based on internal business logic that is not exposed as part of the protocol. For example, in response to an "approval request" message, a service may move to different states based on whether the request is approved or rejected. However, the criteria by which the service moves to this or that state is hidden from the user as it is internal business logic that the provider does not want to expose as part of the protocol definition.

As an example, Figure 1(a) shows a graphical representation of a protocol, called $\mathcal{P}_1$, that describes the external behavior of a store service. Each transition is labeled with a message name followed by the message polarity[2], that

[2] The notion of message polarity is borrowed from [13].

is, whether the message is incoming (plus sign) or outgoing (minus sign). For instance, it specifies that the store service is initially in the Start state, and that clients begin using the service by sending a login message, upon which the service moves to the Logged state (transition (login(+)). We next provide a formal definition of a protocol.

Definition 1. ***(Business protocol)***
A business protocol is a tuple $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ *which consists of the following elements:*

- $\mathcal{S}$ *is a finite set of states.*
- $s_0 \in \mathcal{S}$ *is the initial state.*
- $\mathcal{F} \subseteq \mathcal{S}$ *is a set of final states. If* $\mathcal{F} = \emptyset$*, then* $\mathcal{P}$ *is said to be an empty protocol.*
- $\mathtt{M}$ *is a finite set of messages. For each message* $m \in \mathtt{M}$*, we define a function* $Polarity(\mathcal{P}, m)$ *which will be positive* $(+)$ *if* m *is an input message in* $\mathcal{P}$ *and negative* $(-)$ *if* m *is an output message in* $\mathcal{P}$*. In the sequel, we use the notation* $m(+)$ *(respectively,* $m(-)$*) to denote the polarity of a message* m*.*
- *a finite set* $\mathcal{R} \subseteq \mathcal{S}^2 \times \mathtt{M}$ *of transitions. Each transition* (s, s', m) *identifies a source state* s*, a target state* s' *and either an input or an output message* m *that is either consumed or produced during this transition. In the sequel, we note* $\mathcal{R}(s, s', m)$ *instead of* $(s, s', m) \in \mathcal{R}$*.*

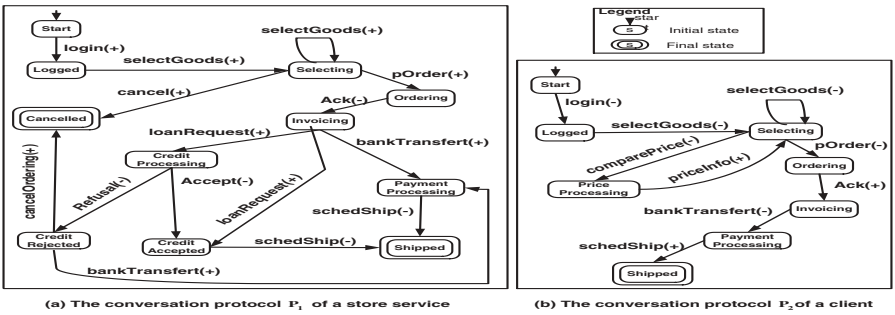

Fig. 1. Business protocols.

2.2 Execution Paths and Execution Trees

In this subsection, we introduce some important concepts and definitions that are used to define the semantics of the protocol model defined above[3]. A protocol defines all the possible conversations that a service supports in terms of alternating sequences of states and messages. We call these sequences *executions paths.* For example, the sequence Start.login(+).Logged.selectGoods(+).Selecting is an execution path of protocol $\mathcal{P}_1$. We are particularly interested in the *complete execution paths* (i.e., paths that start from an initial state and ends at a final state)

[3] See [8] for details on the various process model semantics.

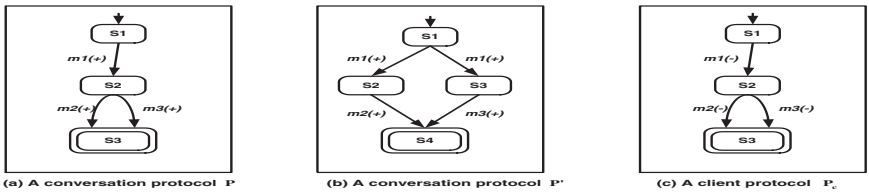

Fig. 2. Comparing protocols with respect to their branching structures.

as they denote the set of correct conversations supported by a service. For example, the execution path Start.login(+).Logged.selectGoods(+).Selecting.cancel(+). Cancelled corresponds to a complete execution path of protocol $\mathcal{P}_1$. The sequence of message exchanges login(+).selectGoods(+).selectGoods(+).cancel(+), extracted from the complete execution path depicted at Figure 3(d), represents a conversation which is *compliant* with (i.e., is allowed by) protocol $\mathcal{P}_1$ of Figure 1(a).

Since protocols are represented using non-deterministic state machines, execution paths are not enough to capture the branching structures of protocols. As an example, Figures 2(a) and (b) show two protocols $\mathcal{P}$ and $\mathcal{P}'$ that specify exactly the same set of compliant conversations (the conversations m1(+).m2(+) and m1(+).m3(+)). However, we can observe that after sending a message m1 in protocol $\mathcal{P}$, a client interacting with $\mathcal{P}$ will have a choice to either send the message m2 or m3, while a client interacting with protocol $\mathcal{P}'$ will not have such a choice. For example, the client protocol $\mathcal{P}_\mathcal{C}$ depicted in Figure 2(c) can interact correctly with the protocol $\mathcal{P}$. However, the interaction of $\mathcal{P}_\mathcal{C}$ with protocol $\mathcal{P}'$ may result in an error (e.g., if $\mathcal{P}_\mathcal{C}$ sends the messages m1 and then m2, while protocol $\mathcal{P}'$ decides to move to the state S3 after receiving the message m1).

To compare protocols with respect to their branching structures, we adopt the well known branching-time approach [8] to describe business protocol semantics. In this approach, the possible conversations allowed by a protocol are characterized in terms of trees, called *execution trees*, instead of paths. The execution trees of a protocol are used to derive what we call *conversation trees*. In a nutshell, conversation trees of a protocol $\mathcal{P}$ capture all the conversations that are compliant with $\mathcal{P}$ (i.e, message exchanges that occur in accordance with the constraints imposed by $\mathcal{P}$) as well as the branching structures of $\mathcal{P}$ (i.e., which messages are allowed at each stage of a conversation).

To formally define the notions of execution and conversation trees, we use the following definition of a tree as in [9]: A tree is a set $\tau \subseteq \mathbb{N}^*$ such that if $xn \in \tau$, for $x \in \mathbb{N}^*$ and $n \in \mathbb{N}$, then $x \in \tau$ and $xm \in \tau$ for all $0 \leq m < n$. The elements of τ represent nodes: the empty word ϵ is the root of τ, and for each node x, the nodes of the form xn, for $n \in \mathbb{N}$, are children of x. Given a pair of set S and M, an $\langle S, M \rangle$-labeled tree is a triple (τ, λ, δ), where τ is a tree, $\lambda : \tau \rightarrow S$ is a node labeling function that maps each node of τ to an element in S, and $\delta : \tau \times \tau \rightarrow M$ is an edge labeling function that maps each edge (x, xn) of τ to an element in M. Then, every path $\rho = \epsilon, n_0, n_0n_1, \ldots$ of τ generates a sequence $\Gamma(\rho) = \lambda(\epsilon).\delta(\epsilon, n_0).\lambda(n_0).\delta(n_0, n_0n_1).\lambda(n_0n_1).\ldots$ of alternating labels from S and M.

Informally, if S and M correspond to the sets of states and messages, we can use an $\langle S, M\rangle$-labeled tree to characterize protocol semantics. In particular, the branches of the tree (once mapped with the labeling functions) represent execution paths, and the tree hierarchy reflects the branching structures of the protocol.

Definition 2. ***(Execution trees and conversation trees)***

Let $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ be a business protocol.

(a) An execution tree of $\mathcal{P}$ is a $\langle \mathcal{S}, \mathtt{M}\rangle$-labeled tree $T = (\tau, \lambda, \delta)$ such that:
- *$\lambda(\epsilon) = s_0$, and*
- *for each edge (x, xn) of τ, we have $\mathcal{R}(\lambda(x), \lambda(xn), \delta(x, xn))$*

An execution tree $T = (\tau, \lambda, \delta)$ is a complete execution tree of the protocol $\mathcal{P}$ if for every leave $x \in \tau$ we have $\lambda(x) \in \mathcal{F}$.

(b) If $T = (\tau, \lambda, \delta)$ is a complete execution tree of a protocol $\mathcal{P}$, then $T^C = (\tau, \lambda^C, \delta)$ where $\lambda^C(x) = \emptyset, \forall x \in \tau$, is a conversation tree which is compliant with protocol $\mathcal{P}$.

For example, Figures 3(a) and (c) show complete execution trees of the protocols $\mathcal{P}$ and $\mathcal{P}'$ of Figure 2. Figure 3(d) shows two complete execution trees which are compliant with the protocol $\mathcal{P}_1$ of Figure 1. Figure 3(b) shows a conversation tree which is compliant with the protocol $\mathcal{P}$ (shown at Figure 2(a)). This conversation tree describes the message exchanges that are accepted by $\mathcal{P}$ (i.e., $\mathsf{m1}(+).\mathsf{m2}(+)$ and $\mathsf{m1}(+).\mathsf{m3}(+)$) as well as the branching choice allowed by $\mathcal{P}$ after receiving the message $\mathsf{m1}$. Conversation trees of a protocol are derived from complete execution trees by removing labels corresponding to the states. For instance, the conversation tree of Figure 3(b) is derived from the complete execution tree of Figure 3(a) by removing the labels of the states $s1$, $s2$, and $s3$. In this paper we use complete execution trees to represent conversations that are compliant with a protocol.

2.3 Protocol Simulation

The notion of *simulation* is used in the literature as a relation to compare labeled transition systems with respect to their branching structures [8, 9]. Simulation is a preorder relation on labeled transition systems that identifies whether a given system has the same branching structures as another one. Here, we introduce a slightly adapted notion of simulation between protocols that will be used to compare protocols with respect to their complete execution trees.

Definition 3. ***(Protocol Simulation)***
Let $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ and $\mathcal{P}' = (\mathcal{S}', s_0', \mathcal{F}', \mathtt{M}', \mathcal{R}')$ be two protocols.

- *A relation $\Gamma \subseteq \mathcal{S} \times \mathcal{S}'$ is a protocol simulation between protocols $\mathcal{P}$ and $\mathcal{P}'$ if whenever $(s_1, s_1') \in \Gamma$ then the following holds:*
  - *$\forall \mathcal{R}(s_1, s_2, m)$ there is an s_2' such that $\mathcal{R}'(s_1', s_2', m)$, $Polarity(\mathcal{P}, m) = Polarity(\mathcal{P}', m)$ and $(s_2, s_2') \in \Gamma$.*
  - *$\forall (s, s') \in \Gamma$, if $s \in \mathcal{F}$ then $s' \in \mathcal{F}'$*

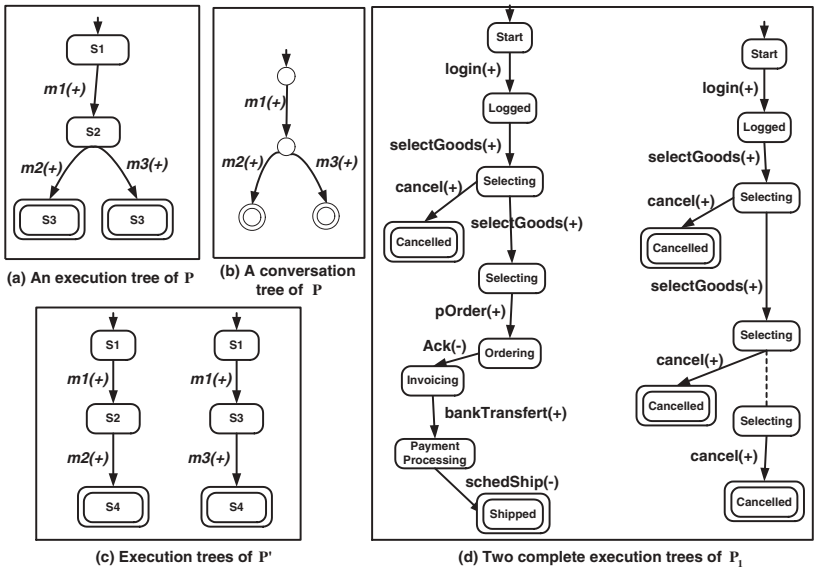

Fig. 3. Example of execution trees of the protocol $\mathcal{P}_1$.

- *We use notation* $s_1 \lesssim s_1'$ *to say that there is a protocol simulation* Γ *such that* $(s_1, s_1') \in \Gamma$.
- *We say that protocol* $\mathcal{P}$ *is simulated by* $\mathcal{P}'$ *(noted* $\mathcal{P} \lesssim \mathcal{P}'$*) iff* $s_0 \lesssim s_0'$.
- *We say that two protocols* $\mathcal{P}$ *and* $\mathcal{P}'$ *are* similar *(noted* $\mathcal{P} \cong \mathcal{P}'$*) iff* $\mathcal{P} \lesssim \mathcal{P}'$ *and* $\mathcal{P}' \lesssim \mathcal{P}$.

The following lemma[4] states that the simulation relation allows to compare protocols with respect to their complete execution trees.

Lemma 1. *Let* $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ *and* $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$ *be two protocols.*

(i) $\mathcal{P}_1 \lesssim \mathcal{P}_2$ *iff there exists a node labeling function* $\lambda^2 : \tau \rightarrow \mathcal{S}^2$ *such that for each complete execution tree* $T = (\tau, \lambda, \delta)$ *of* $\mathcal{P}_1$, $T' = (\tau, \lambda^2, \delta)$ *is a complete execution tree of* $\mathcal{P}_2$ *and* $Polarity(\mathcal{P}_1, \delta(x, x_n)) = Polarity(\mathcal{P}_2, \delta(x, x_n))$ *for each edge* (x, x_n) *of* τ.

(ii) $\mathcal{P}_1 \cong \mathcal{P}_2$ *iff* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *have exactly the same set of complete execution trees, modulo the name of the states.*

2.4 Protocol Interactions

In the previous subsections we focused on representing a protocol supported by a given service and comparing two service protocols using the simulation relation. We now address the joint analysis of two protocols, that of a requestor and that

[4] It should be noted that lemma proofs are not presented due to space reasons.

of a provider, to see if interactions between them are compatible. By defining the constraints on the ordering of the messages that a web service accepts, a business protocol makes explicit to clients how they can *correctly* interact with the service (i.e., without generating errors due to incorrect sequencing of messages)[5] [13, 3]. For example, a service that supports the reversed protocol $\bar{\mathcal{P}}_1$ obtained from $\mathcal{P}_1$ of Figure 1(a) by reversing the direction of the messages (i.e., input messages becomes outputs and vice versa) can interact correctly with the store service. Interactions between two given protocols can also be characterized in terms of execution paths and trees.

As an example, consider again the protocol $\mathcal{P}_1$ depicted in Figure 1(a) and its reversed protocol $\bar{\mathcal{P}}_1$. As the two protocols have exactly the same states, if s is a state in the protocol $\mathcal{P}_1$, we use $\bar{s}$ to denote the corresponding state in the protocol $\bar{\mathcal{P}}_1$. The path $(\texttt{Start}, \bar{\texttt{Start}}).\texttt{login}.(\texttt{Logged}, \bar{\texttt{Logged}})$ corresponds to a possible interaction between protocols $\mathcal{P}_1$ and $\bar{\mathcal{P}}_1$. This path indicates that, at the beginning, the two protocols $\mathcal{P}_1$ and $\bar{\mathcal{P}}_1$ are respectively at the states `Start` and $\bar{\texttt{Start}}$. Then, protocol $\bar{\mathcal{P}}_1$ sends message `login` and goes to state $\bar{\texttt{Logged}}$ while protocol $\mathcal{P}_1$ receives message `login` and goes to state `Logged`. The path $(\texttt{Start}, \bar{\texttt{Start}}).\texttt{login}.(\texttt{Logged}, \bar{\texttt{Logged}})$ is called an *interaction path* of protocols $\mathcal{P}_1$ and $\bar{\mathcal{P}}_1$. Each state in this interaction path consists of a state of $\mathcal{P}_1$ together with a state of $\bar{\mathcal{P}}_1$. The transition `login` indicates that an input `login` message of one of the protocols coincides with an output `login` message of the other protocol. Consequently, the polarity of the messages that appear in an interaction path is not defined.

Correct interactions between two protocols are captured by using the notion of *complete interaction trees*, i.e., interaction trees in which both protocols start at an initial state and end at a final state. For example, the complete interaction tree of Figure 4(a) describes a possible correct interaction between the protocols $\mathcal{P}_1$ of Figure 1(a) and its reversed protocol $\bar{\mathcal{P}}_1$. The notion of *interaction tree* is formally defined below.

Definition 4. ***(interaction tree)*** *An interaction tree between two protocols* $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ *and* $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$ *is a* $\langle(\mathcal{S}^1 \times \mathcal{S}^2, \mathtt{M}^1 \cap \mathtt{M}^2)\rangle$*-labeled tree* $I = (\tau, \lambda, \delta)$ *such that:*

- $\lambda(\epsilon) = (s_0^1, s_0^2)$*, and*
- *For* $x \in \tau$*,* $\lambda(x) = (s_x^1, s_x^2)$ *such that* $s_x^1 \in \mathcal{S}^1$ *and* $s_x^2 \in \mathcal{S}^2$*. Then, for each edge* (x, xn) *of* τ*, we have:* $\mathcal{R}^1(s_x^1, s_{xn}^1, \delta(x, xn))$ *and* $\mathcal{R}^2(s_x^2, s_{xn}^2, \delta(x, xn))$*, and* $Polarity(\mathcal{P}_1, \delta(x, xn)) \neq Polarity(\mathcal{P}_2, \delta(x, xn))$
  An interaction tree $I = (\tau, \lambda, \delta)$ *is a complete interaction tree of the protocols* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *if for every leave* $x \in \tau$ *we have* $\lambda(x) \in \mathcal{F}^1 \times \mathcal{F}^2$*.*

In the sequel, an interaction between two protocols is characterized by the set of the complete interaction trees of these protocols.

It should be noted that the notions of simulation and interactions defined above focus on comparing protocols based on their structure and their messages,

[5] Recall that structural and semantics interoperability [6] are outside the scope of this paper.

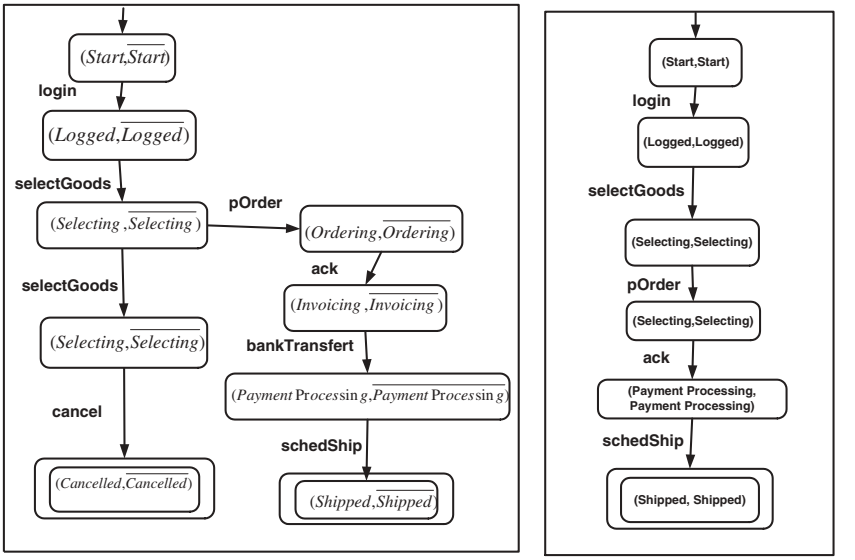

Fig. 4. Interaction trees.

regardless of how states are named. Specifically, when in the formal definition we place conditions on the two protocols having the same message (with the same or opposite polarity), we mean that they have to refer to the same WSDL message, as defined by its fully qualified name. Naming of the states is instead irrelevant, as it has no effect on identifying the conversations allowed by a protocol.

3 Protocol Management Operators

To assess commonalities and differences between protocols, as well as whether two protocols can interact with each other, we define a set of generic operators to manipulate business protocols, namely: *compatible composition* of protocols, *intersection* of protocols, *difference* between protocols, and *projection* of protocol on a given role. The proposed operators take protocols as their operands and return a protocol as their result. Although the proposed operators are generic in the sense that they can be useful in several tasks related to management and analysis of business protocols, we will show in the next section how these operators can be used for analysing protocols compatibility and replaceability. Effecient algorithms that implement the proposed operators as well as correctness proofs are given in [4].

3.1 Compatible Composition

The operator *compatible composition* allows to characterize possible interactions between two protocols, that of a requestor and that of a provider (i.e., the result-

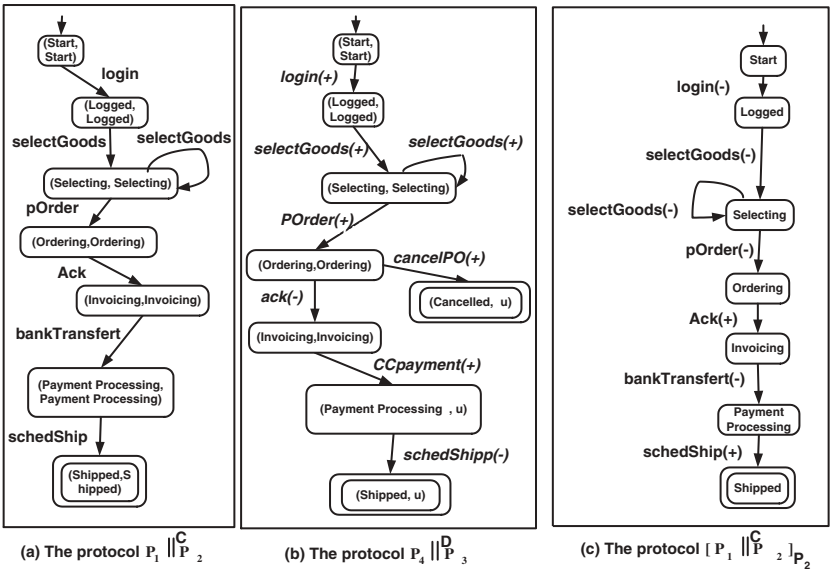

Fig. 5. Examples of protocol management operators.

ing protocol describes all the interaction trees between the considered protocols, and therefore characterizes the possible conversations that can take place between the requestor and the provider). This operator, denoted as $||^{\mathtt{C}}$, takes as input two business protocols and returns a protocol, called a *compatible composition protocol*, that describes the set of complete interactions trees between the input protocols. Informally, the initial state of the resulting protocol is obtained by combining the initial states of the input protocols, final states are obtained by combining the final states of the input protocols, while intermediate states are constructed by combining the intermediate states of the input protocols. The resulting protocol is constructed by considering messages of the two input protocols which have same names but opposite polarities, and that allow execution paths to flow from the start state to end states of the new protocol. All the states that are not reachable from the initial state of the resulting protocol as well as the states that cannot lead to a final state are removed from the resulting protocol. If the result of a compatible composition of two protocols is empty, this means that no conversation is possible between two services that support these protocols. Otherwise, the result is the identification of possible interactions between these protocols.

As an example, Figure 5(a) shows protocol $\mathcal{P}_1||^{\mathtt{C}}\mathcal{P}_2$ that describes all the possible complete interaction trees between protocols $\mathcal{P}_1$ of Figure 1(a) and $\mathcal{P}_2$ of Figure 1(b).

Definition 5. *(Compatible composition)*
Let $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ *and* $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$ *be two protocols. The compatible composition* $\mathcal{P} = \mathcal{P}_1||^{\mathtt{C}}\mathcal{P}_2$ *is a protocol* $(\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ *where:*

- $\mathcal{S} \subseteq \mathcal{S}^1 \times \mathcal{S}^2$ *is a finite set of states,*
- $s_0 = (s_0^1, s_0^2)$ *is the initial state,*
- $\mathcal{F} \subseteq \mathcal{F}^1 \times \mathcal{F}^2$ *is a set of final states,*
- $\mathtt{M} \subseteq \mathtt{M}^1 \cap \mathtt{M}^2$ *is a set of messages. Note that, the polarity function is not defined for the messages in an a compatible composition protocol.*
- $\mathcal{R}((s^1, s^2), (q^1, q^2), m)$ *iff* $\mathcal{R}^1(s^1, q^1, m)$ *and* $\mathcal{R}^2(s^2, q^2, m)$ *and* $Polarity(\mathcal{P}_1, m) \neq Polarity(\mathcal{P}_2, m)$.
- $\forall (s^1, s^2) \in \mathcal{S}^1 \times \mathcal{S}^2$, *the state* $(s^1, s^2) \in \mathcal{S}$ *iff* (s^1, s^2) *belongs to a complete execution path of* $\mathcal{P}$ *(i.e., a path that goes from the initial state* (s_0^1, s_0^2) *to a final state* $(s_i^1, s_j^2) \in \mathcal{F}$*).*

3.2 Intersection

The intersection operator allows the computation of the largest common part between two protocols. The intersection operator, denoted as $||^{\mathtt{I}}$, takes as input two business protocols and returns a protocol that describes the set of complete execution trees that are common between the two input protocols. The resulting protocol is called an *intersection protocol*. This operator combines the two input protocols as follows: states of the resulting protocols are constructed using the same procedure as in the compatible composition operator. However, unlike compatible composition, the intersection protocol is constructed by considering messages of the input protocols which have same names and polarities.

Definition 6. *(Intersection)*
Let $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ *and* $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$ *be two protocols. The intersection* $\mathcal{P} = \mathcal{P}_1 ||^{\mathtt{I}} \mathcal{P}_2$ *is a protocol* $(\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ *where:*

- $\mathcal{S} \subseteq \mathcal{S}^1 \times \mathcal{S}^2$,
- $s_0 = (s_0^1, s_0^2)$,
- $\mathcal{F} \subseteq \mathcal{F}^1 \times \mathcal{F}^2$,
- $\mathtt{M} \subseteq \mathtt{M}^1 \cap \mathtt{M}^2$.
- $\mathcal{R}((s, q), (s', q'), m)$ *iff* $\mathcal{R}^1(s, s', m)$ *and* $\mathcal{R}^2(q, q', m)$ *and* $Polarity(\mathcal{P}_1, m) = Polarity(\mathcal{P}_2, m)$.
- $\forall (s^1, s^2) \in \mathcal{S}^1 \times \mathcal{S}^2$, *the state* $(s^1, s^2) \in \mathcal{S}$ *iff* (s^1, s^2) *belongs to a complete execution path of* $\mathcal{P}$ *(i.e., a path that goes from the initial state* (s_0^1, s_0^2) *to a final state* $(s_i^1, s_j^2) \in \mathcal{F}$*).*

Note that the intersection protocol preserves the polarity of the messages (i.e., $\forall m \in \mathtt{M}, Polarity(\mathcal{P}_1 ||^{\mathtt{I}} \mathcal{P}_2, m) = Polarity(\mathcal{P}_1, m) = Polarity(\mathcal{P}_2, m)$).

3.3 Difference

While the intersection identifies common aspects between two protocols, the *difference operator*, denoted as $||^{\mathtt{D}}$, emphasizes their differences. This operator takes as input two protocols $\mathcal{P}_1$ and $\mathcal{P}_2$, and returns a protocol called *difference protocol*, whose purpose is to describe the set of all complete execution trees of $\mathcal{P}_1$ that are not common with $\mathcal{P}_2$. As shown below, we compute the difference as a protocol where states are combination of states of $\mathcal{P}_1$ and $\mathcal{P}_2$, as opposed to deriving

the subset of $\mathcal{P}_1$ that is not part of $\mathcal{P}_2$. This will allow us to reuse procedures similar to those developed for computing results of previous operators.

Definition 7. *(Difference)*
Let $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ *and* $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$ *be two protocols and let* $\mu \notin \mathcal{S}^1 \cup \mathcal{S}^2$ *be a new state name. The difference* $\mathcal{P} = \mathcal{P}_1 ||^{\mathtt{D}} \mathcal{P}_2$ *is a protocol* $(\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ *where:*

- $\mathcal{S} \subseteq \mathcal{S}^1 \times (\mathcal{S}^2 \cup \{\mu\})$,
- $s_0 = (s_0^1, s_0^2)$,
- $\mathcal{F} \subseteq \mathcal{F}^1 \times ((\mathcal{S}^2 \cup \{\mu\}) \setminus \mathcal{F}^2)$,
- $\mathtt{M} \subseteq \mathtt{M}^1$.
- $\mathcal{R}((s,q),(s',q'),m)$, *with* $q, q' \in \mathcal{S}^2$, *iff* $\mathcal{R}^1(s,s',m)$, $\mathcal{R}^2(q,q',m)$ *and* $Polarity(\mathcal{P}_1, m) = Polarity(\mathcal{P}_2, m)$,
- $\mathcal{R}((s,q),(s',\mu),m)$, *with* $q \in \mathcal{S}^2$, *iff* $\mathcal{R}^1(s,s',m)$ *and not exists* $q' \in \mathcal{S}^2$ *such that* $\mathcal{R}^2(q,q',m)$ *and* $Polarity(\mathcal{P}_1, m) = Polarity(\mathcal{P}_2, m)$,
- $\mathcal{R}((s,\mu),(s',\mu),m)$ *iff* $\mathcal{R}^1(s,s',m)$.
- $\forall (s^1, s^2) \in \mathcal{S}^1 \times \mathcal{S}^2$, *the state* $(s^1, s^2) \in \mathcal{S}$ *iff* (s^1, s^2) *belongs to a complete execution path of* $\mathcal{P}$ *(i.e., a path that goes from the initial state* (s_0^1, s_0^2) *to a final state* $(s_i^1, s_j^2) \in \mathcal{F}$*).*

As an example, Figure 5(b) shows the difference protocol $\mathcal{P}_4 ||^{\mathtt{D}} \mathcal{P}_3$ that describes all the complete execution trees of the protocol $\mathcal{P}_4$ of Figure 6(b) that are not allowed by the protocol $\mathcal{P}_3$ of Figure 6(a). From this, it can be inferred, e.g, that the sequence of messages login(+).selectGoods(+).POrder(+).cancelPO(+), which is derived from a complete execution path of the difference protocol, is a conversation which is allowed by the protocol $\mathcal{P}_4$ but it is not allowed by $\mathcal{P}_3$.

3.4 Projection

In this section, we discuss the projection of a protocol obtained by using one of the previous operators (i.e, compatible composition, intersection, or difference) on a participant protocol. In the case of compatible composition, the projection of $\mathcal{P}_1 ||^{\mathtt{C}} \mathcal{P}_2$ on the protocol $\mathcal{P}_1$, denoted as $[\mathcal{P}_1 ||^{\mathtt{C}} \mathcal{P}_2]_{\mathcal{P}_1}$, allows to identify the part of the protocol $\mathcal{P}_1$ that is able to interact correctly with the protocol $\mathcal{P}_2$. While a compatible composition protocol $\mathcal{P}$ allows to characterize the possible interactions between two business protocols each defining the behavior of a service playing a certain *role* in a collaboration, the projection of $\mathcal{P}$ allows the extraction of a protocol that defines the role a service plays in a collaboration (e.g., a customer) defined by $\mathcal{P}$. This is very important since the expected behavior of a service in a collaboration constitutes an important part of the requirements for the implementation.

As an example, Figure 5(c) shows the projection of protocol $\mathcal{P}_1 ||^{\mathtt{C}} \mathcal{P}_2$ of Figure 5(a) on protocol $\mathcal{P}_2$. The obtained protocol describes the part of protocol $\mathcal{P}_2$ that can be used to interact correctly with $\mathcal{P}_1$ (i.e, the role of $\mathcal{P}_2$ in $\mathcal{P}_1 ||^{\mathtt{C}} \mathcal{P}_2$).

Briefly stated, the projection of a protocol obtained using an intersection or a difference is defined as follows. In the case of intersection, the projection

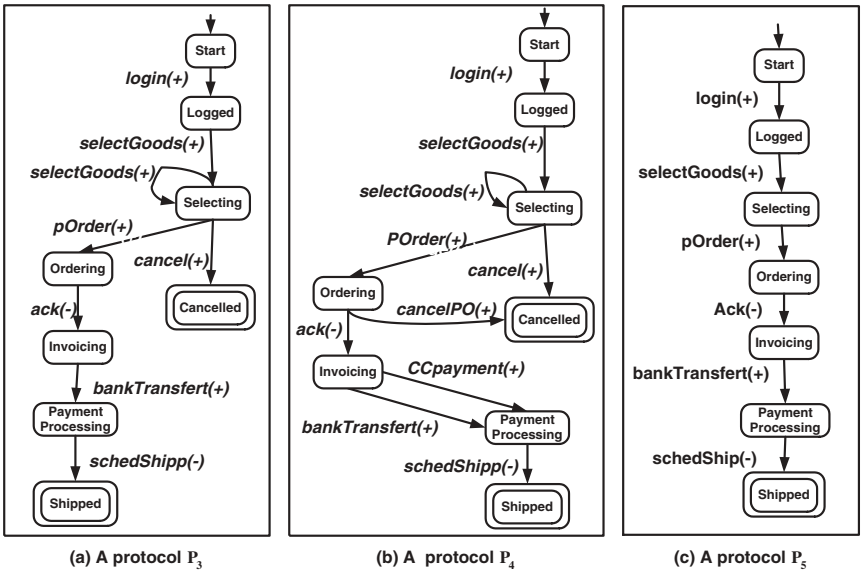

Fig. 6. Business protocols of two other store services.

$[\mathcal{P}_1||^{\mathtt{I}}\mathcal{P}_2]_{\mathcal{P}_1}$ allows to identify the part of the protocol $\mathcal{P}_1$ that is common with the protocol $\mathcal{P}_2$. In the case of difference, the projection $[\mathcal{P}_1||^{\mathtt{D}}\mathcal{P}_2]_{\mathcal{P}_1}$ allows to identify the part of the protocol $\mathcal{P}_1$ that is not supported by the protocol $\mathcal{P}_2$. Below, we give the formal definition of the projection of a protocol obtained by using compatible composition, intersection, or difference.

Definition 8. *(Projection) Let* $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ *be a protocol obtained using compatible composition, intersection, or difference of two protocols* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *(e.g.,* $\mathcal{P} = \mathcal{P}_1||^{\mathtt{C}}\mathcal{P}_2$*). A projection of* $\mathcal{P}$ *on the protocol* $\mathcal{P}_1$*, denoted as* $[\mathcal{P}]_{\mathcal{P}_1}$*, is a protocol* $(\mathcal{S}', s'_0, \mathcal{F}', \mathtt{M}, \mathcal{R})$ *obtained from* $\mathcal{P}$ *by projecting the states of* $\mathcal{P}$ *on* $\mathcal{P}_1$ *(i.e., replacing each state* $(s_i^1, s_j^2) \in \mathcal{S}$ *by the state* s_i^1 *in* $\mathcal{S}'$*) and by defining the polarity function of the messages as follows:* $Polarity([\mathcal{P}]_{\mathcal{P}_1}, m) = Polarity(\mathcal{P}_1, m), \forall m \in \mathtt{M}$.

4 Taxonomy of Protocols Compatibility and Replaceability

This section analyzes service protocols compatibility and replaceability. Service compatibility refers to capabilities of services to interoperate while service replaceability refers to the ability of a given service to be used instead of another service, in such a way that the change is transparent to external clients. We define and characterize several types of protocols compatibility (respectively, replaceability). We show how, given two services and the corresponding protocols, it is possible to identify their compatibility (respectively, replaceability) levels

using the operators introduced in section 3. Instead of simple black and white compatibility and replaceability measures (i.e, whether two services are compatible or not, whether a service can replace another or not), we propose to consider different classes of protocols compatibility and replaceability.

4.1 Compatibility Classes

We identify two classes of protocols compatibility which provide basic building blocks for analysing complex interactions between service protocols.

- *Partial compatibility* (or simply, compatibility): A protocol $\mathcal{P}_1$ is partially compatible with another protocol $\mathcal{P}_2$ if there are some executions of P1 that can interoperate with P2, i.e., if there is at least one possible conversation that can take place among two services supporting these protocols
- *Full compatibility*: a protocol $\mathcal{P}_1$ is fully compatible with another protocol $\mathcal{P}_2$, if all the executions of $\mathcal{P}_1$ can interoperate with $\mathcal{P}_2$, i.e., any conversation that can be generated by $\mathcal{P}_1$ is understood by $\mathcal{P}_2$.

These notions of compatibility are very useful in the context of Web services. For example, it does not make sense to have interactions with services for which there is no (partial or total) compatibility, as no meaningful conversation can be carried on. Furthermore, if there is only partial compatibility, the developer and the Web service middleware need to be aware of this, as the service will not be able to exploit its full capabilities when interacting with the partially compatible one: indeed, in this case, it is not sufficient that a service implementation is compliant with its advertised protocol, as additional constraints are posed by the fact that the service is interacting with another one whose protocols is only partially compatible, and hence some conversations are disallowed.

As an example, Protocol $\mathcal{P}_1$ of Figure 1(a) can interact with the its reversed protocol $\bar{\mathcal{P}}_1$ without generating errors and, hence, $\mathcal{P}_1$ is *fully compatible* with $\bar{\mathcal{P}}_1$. However, this is not the case for the protocol $\mathcal{P}_1$ of Figure 1(a) and the protocol $\mathcal{P}_2$ of Figure 1(b). When the protocol $\mathcal{P}_2$ is at the state Selecting, it can send a message comparePrice, e.g., to look for the best price of a given product, and goes into a state Price Processing where it waits for an input message priceInfo. These two transitions do not coincide with the transitions of the protocol $\mathcal{P}_1$ (i.e., the protocol $\mathcal{P}_1$ does not accept an input message comparePrice at the state Selecting nor it is able to generate an output message priceInfo). Clearly, there are some executions of the protocol $\mathcal{P}_2$ that cannot interact with the protocol $\mathcal{P}_1$. However, we can observe that there are some cases where the protocol $\mathcal{P}_2$ is able to interact correctly with the protocol $\mathcal{P}_1$ (i.e., there are some executions of $\mathcal{P}_2$ that are compatible with executions of the protocol $\mathcal{P}_1$). An example of such an interaction is given by the complete interaction tree between $\mathcal{P}_1$ and $\mathcal{P}_2$ depicted at Figure 4(b). Hence, the protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ are(partially) compatible.

We use the boolean operator $P\text{-}compat(\mathcal{P}_1, \mathcal{P}_2)$ (respectively, $F\text{-}compat(\mathcal{P}_1, \mathcal{P}_2)$) to test if the protocol $\mathcal{P}_1$ is partially compatible (respectively, fully compatible) with the protocol $\mathcal{P}_2$. The following lemma gives necessary and sufficient conditions to identify the compatibility level between two protocols.

Lemma 2. *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two business protocols, then*

(i) P-compat$(\mathcal{P}_1, \mathcal{P}_2)$ iff $\mathcal{P}_1 ||^{\mathsf{C}} \mathcal{P}_2$ is not an empty protocol (i.e., the set of its final states is not empty).
(ii) F-compat$(\mathcal{P}_1, \mathcal{P}_2)$ iff $[\mathcal{P}_1 ||^{\mathsf{C}} \mathcal{P}_2]_{\mathcal{P}_1} \cong \mathcal{P}_1$

Note that *full compatibility* is not a symmetric relation, i.e., *F-compat*$(\mathcal{P}_1, \mathcal{P}_2)$ does not imply *F-compat*$(\mathcal{P}_2, \mathcal{P}_1)$ (however, *F-compat*$(\mathcal{P}_1, \mathcal{P}_2)$ implies *P-compat*$(\mathcal{P}_2, \mathcal{P}_1)$).

4.2 Replaceability Classes

Repleacability analysis helps us identify if we can use a service supporting a certain protocol in place of a service supporting a different protocol, both in general and when interacting with a certain client. It also helps developer to manage service evolution, as when a service is modified there is the need for understanding if it can still support all conversations the previous version supported. We identify four replaceability classes between protocols, namely: equivalence, subsumption, replaceability with respect to a client protocol and replaceability with respect to an interaction role. These replaceability classes provide basic building blocks for analysing the commonalities and differences between service protocols.

1. *Protocols equivalence*: two business protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent if they are mutually substituable, i.e., the two protocols can be interchangeably used in any context and the change is transparent to clients. We use the boolean operator $Equiv(\mathcal{P}_1, \mathcal{P}_2)$ to test the equivalence of protocols $\mathcal{P}_1$ and $\mathcal{P}_2$.
2. *Protocol subsumption*: a protocol $\mathcal{P}_1$ is subsumed by another protocol $\mathcal{P}_2$, if the externally visible behavior of $\mathcal{P}_1$ encompasses the externally visible behavior of $\mathcal{P}_2$, i.e., if $\mathcal{P}_1$ supports at least all the conversations that $\mathcal{P}_2$ supports. In this case, protocol $\mathcal{P}_1$ can be transparently used instead of $\mathcal{P}_2$ but the opposite is not necessarily true. We use the boolean operator $Subs(\mathcal{P}_1, \mathcal{P}_2)$ to test if $\mathcal{P}_2$ subsumes $\mathcal{P}_1$. It should be noted that equivalence is stronger than subsumption (i.e., $Equiv(\mathcal{P}_1, \mathcal{P}_2)$ implies $Subs(\mathcal{P}_1, \mathcal{P}_2)$ and $Subs(\mathcal{P}_2, \mathcal{P}_1)$). The protocol $\mathcal{P}_1$ of Figure 1(a) is subsumed by the protocol $\mathcal{P}_3$ of Figure 6(a).
3. *Protocol replaceability with respect to a client protocol*: The previous definitions discussed replaceability in general. However, it may be important to understand if a service can be used to replace another one when interacting with a certain client. This leads to a weaker definition of replaceability: a protocol $\mathcal{P}_1$ can replace another protocol $\mathcal{P}_2$ with respect to a client protocol Pc, denoted as $Repl_{[\mathcal{P}_c]}(\mathcal{P}_1, \mathcal{P}_2)$, if $\mathcal{P}_1$ behaves similarly as $\mathcal{P}_2$ when interacting with a specific protocol Pc. Hence, if $Repl_{[\mathcal{P}_c]}(\mathcal{P}_1, \mathcal{P}_2)$ than $\mathcal{P}_1$ can replace $\mathcal{P}_2$ to interact with $\mathcal{P}_c$ and this change is transparent to the client $\mathcal{P}_c$. It should be noted that, this replaceability class is also weaker than subsumption (i.e., $Subs(\mathcal{P}_1, \mathcal{P}_2)$ implies $Repl_{[\mathcal{P}_c]}(\mathcal{P}_1, \mathcal{P}_2)$ for any protocol P_c). For example, Protocol $\mathcal{P}_1$ of Figure 1(a) is not subsumed by protocol $\mathcal{P}_4$ of Figure 6(b), as $\mathcal{P}_4$ allows a client to cancel an order (message $\mathsf{cancelPO}(+)$ at

state Ordering) and also accept payments by credit card (message CCpayment(-) at state Invoicing) while $\mathcal{P}_1$ does not support these two possibilities. Hence, $\mathcal{P}_1$ cannot replace $\mathcal{P}_4$ for arbitrary clients. However, we can observe that $\mathcal{P}_1$ can replace $\mathcal{P}_4$ when interacting with the protocol $\mathcal{P}_2$ of Figure 1(b) as this client never cancel an ordering and it also always performs its payments by bank transfer.

4. *Protocol replaceability with respect to an interaction role*: Let $\mathcal{P}_R$ be a business protocol. A protocol $\mathcal{P}_1$ can replace another protocol $\mathcal{P}_2$ with respect to a role $\mathcal{P}_R$, denoted as $Repl\_Role_{[\mathcal{P}_R]}(\mathcal{P}_1, \mathcal{P}_2)$, if $\mathcal{P}_1$ behaves similarly as $\mathcal{P}_2$ when $\mathcal{P}_2$ behaves as $\mathcal{P}_R$. This replaceability class allows to identify executions of a protocol $\mathcal{P}_2$ that can be replaced by the protocol $\mathcal{P}_1$ even in the case when $\mathcal{P}_1$ and $\mathcal{P}_2$ are not comparable with respect to any of the previous replaceability classes. The class $Repl\_Role_{[\mathcal{P}_R]}(\mathcal{P}_1, \mathcal{P}_2)$ is the weakest replaceability class (i.e., $Repl_{[\mathcal{P}_R]}(\mathcal{P}_1, \mathcal{P}_2)$ implies $Repl\_Role_{[\mathcal{P}_R]}(\mathcal{P}_1, \mathcal{P}_2)$). For example, consider the protocol $\mathcal{P}_2$ of Figure 1(b) and its reversed protocol $\bar{\mathcal{P}}_2$. Protocol $\mathcal{P}_4$ of Figure 6(b) cannot replace protocol $\bar{\mathcal{P}}_2$ when interacting with client $\mathcal{P}_2$ (i.e., $Repl_{[\mathcal{P}_2]}(\mathcal{P}_4, \bar{\mathcal{P}}_2)$ does not hold). This is because protocol $\mathcal{P}_4$ does not accept an input message comparePrice at the state Selecting. However, even in this case, a client $\mathcal{P}_2$ may be interested to know for which executions it can use $\mathcal{P}_4$ instead of $\bar{\mathcal{P}}_2$. For example, we can observe that $\mathcal{P}_4$ can replace $\bar{\mathcal{P}}_2$ in all the interactions in which $\bar{\mathcal{P}}_2$ behaves as the protocol $\mathcal{P}_5$ of Figure 6(c) (i.e., $Repl\_Role_{[\mathcal{P}_5]}(\mathcal{P}_4, \bar{\mathcal{P}}_2)$). In other words, the protocol $\mathcal{P}_5$ exhibits to a given client executions of $\bar{\mathcal{P}}_2$ for which it is possible to use $\mathcal{P}_4$ instead of $\bar{\mathcal{P}}_2$.

The following lemma characterizes the replaceability levels of two given protocols using the operators introduced in previous sections.

Lemma 3. *Let $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_c$ and $\mathcal{P}_R$ be business protocols.*

1. $Equiv(\mathcal{P}_1, \mathcal{P}_2)$ *iff* $\mathcal{P}_1 \cong \mathcal{P}_2$
2. $Subs(\mathcal{P}_1, \mathcal{P}_2)$ *iff* $\mathcal{P}_2 \lesssim \mathcal{P}_1$
3. $Repl_{[\mathcal{P}_c]}(\mathcal{P}_1, \mathcal{P}_2)$ *iff* $[\mathcal{P}_c||^{\mathsf{C}}\mathcal{P}_2]_{\mathcal{P}_2} \lesssim \mathcal{P}_1$ *(or equivalently, iff* $\mathcal{P}_c||^{\mathsf{C}}[\mathcal{P}_2||^{\mathsf{D}}\mathcal{P}_1]_{\mathcal{P}_2}$ *is an empty protocol)*
4. $Repl\_Role_{[\mathcal{P}_R]}(\mathcal{P}_1, \mathcal{P}_2)$ *iff* $P_R \lesssim [\mathcal{P}_1||^{\mathsf{I}}\mathcal{P}_2]_{\mathcal{P}_1}$.

Note that this lemma provides two equivalent characterizations of class $Repl_{[\mathcal{P}_c]}(\mathcal{P}_1, \mathcal{P}_2)$ (item 3 of the lemma). The second characterization (i.e., $\mathcal{P}_c||^{\mathsf{C}}[\mathcal{P}_1||^{\mathsf{D}}\mathcal{P}_2]_{\mathcal{P}}$) can be useful to check whether $\mathcal{P}_2$ can be used instead of $\mathcal{P}_1$ with respect to a client $\mathcal{P}_c$ in those cases where the protocol $\mathcal{P}_2$ is not fully accessible (e.g., $\mathcal{P}_2$ is hidden for security reasons). Furthermore, such a characterization may be interesting for change support as it allows to incrementally check whether a given client protocol $\mathcal{P}_c$ used to interact with a protocol $\mathcal{P}_1$ can still interact correctly with a new version $\mathcal{P}_2$ of the protocol $\mathcal{P}_1$.

5 Discussion

We believe that the effective use and widespread adoption of Web service technologies and standards requires: (i) high-level frameworks and methodologies for

supporting automated development and interoperability (e.g., code generation, compatibility), and (ii) identification of appropriate abstractions and notations for specifying service requirements and characteristics. Service protocol management as proposed in this paper offers a set of mechanisms for the automation of services development and interoperability.

Several efforts recognize aspects of protocol specifications in component-based models [7, 13]. These efforts provide models (e.g., pi-calculus -based languages for component interface specifications) and algorithms (e.g., compatibility checking) that can be generalized for use in Web service protocol specifications and management. Indeed, various efforts in the general area of formalizing Web service description and composition languages emerged recently [5, 11]. However, in terms of managing the Web service development life-cycle, technology is still in the early stages. The main contribution of the work presented in this paper is a framework that leverages emerging Web services technologies and established modeling notation (state machine-based formalism) to provide high-level support for analyzing degrees of commonalities and differences between protocols as well interoperation possibilities of interacting Web services. In the following we briefly discuss how the framework presented in this paper can be leveraged to better support the service lifecycle management.

Development-time support. During service development, protocol analysis can assist in assessing the compatibility of the newly created service (and service protocol) with the other services with which it needs to interact. The protocol analysis will in particular help users to identify which part of the protocol are compatible and which are not, therefore suggesting possible areas of modifications that we need to tackle if we want to increase the level of compatibility with a desired service.

Runtime support. In terms of runtime support, the main application of compatibility analysis is in dynamic binding. In fact, just like for static binding, the benefit of protocol analysis is that search engines can restrict the services they return to those that are compatible. This is essential, as there is no point in returning services that are not protocol-compatible, since no interoperation will be possible (unless there is a mediation mechanism that can make interaction possible, as discussed below).

Change Support. Web services operate autonomously within potentially dynamic environments. In particular, component or partner services may change their protocols, others may become unavailable, and still others may emerge. Consequently, services may fail to invoke required operations when needed. The proposed operators allow to statically and dynamically identify alternative services based on behavior equivalence and substitution. Protocols analysis and management provide also opportunities to: (i) help understanding mismatch between protocols, (ii) help understand if a new version of a service protocol is compatible with the intended clients, and the like.

The framework presented in this paper is one of the components of a broader CASE tool, partially implemented, that manages the entire service development

lifecycle [2, 3, 10]. In this paper, we focused on analysing and managing Web service protocols. Another component (presented in [2]) of this framework features a generative approach where conversation protocols are specified using an extended state machine model, composition models are specified using statecharts, and executable processes are described using BPEL. Through this component, users can visually edit service conversation and composition models and automatically generate the BPEL skeletons, which can then be extended by the developers and eventually executed using BPEL execution engine such as the IBM's BPWS4J (www.alphaworks.ibm.com/tech/bpws4j). We are also considering the extension of the proposed approach to cater for trust negotiation and security protocols in Web services, by exploring the leverages between conversation and trust negotiation protocols [10], that can both be specified through state machines, although at different levels of abstractions. The proposed framework supports also lifecycle management of trust negotiation protocols [10]. We introduced a set of change operations that are used to modify protocol specifications. Strategies are presented to allow migration of ongoing negotiations to a new protocol specification. Details about these features of framework can be found in [2, 10]. Finally, our current research also includes addressing the problem of designing and testing for compatibility, trying in particular to understand how to develop test cases that can test that two services can interact and how to devise a methodology for service development that facilitates protocol compatibility.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications*. Springer Verlag, 2004.
2. K. Baina, B. Benatallah, F. Casati, and F. Toumani. Model-Driven Web Service Development. In *CAiSE'04*, volume 3084 of *LNCS*, Riga, Latvia, 2004. Springer.
3. B. Benatallah, F. Casati, and F. Toumani. Web Service Conversation Modeling: A Cornerstone for e-Business Automation. *IEEE Internet Computing*, 6(1), 2004.
4. B. Benatallah, F.Casati, and F. Toumani. A Framework for Modeling, Analyzing, and Managing Web Service Protocols. Technical Report 0430, CSE-UNSW, August, 2004.
5. T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW'03*. ACM, 2003.
6. C. Bussler. *B2B Integration: Concepts and Architecture*. Springer Verlag, 2003.
7. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding Roles to CORBA Objects. *IEEE TSE*, 29(3):242–260, March 2003.
8. R.J. van Glabbeek. The Linear Time – Branching Time Spectrum (extended abstract). In *CONCUR'90*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.
9. T.A. Henzinger, S. Qadeer, S.K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. *ACM Trans. Prog. Lang. Syst.*, 24(1):51–64, Jan' 02.
10. H.Skogsrud, B.Benatallah, and F.Casati. Model-Driven Trust Negotiation for Web Services. *IEEE Internet Computing*, 7(6), October 2003.
11. M. Mecella, B. Pernici, and P. Craca. Compatibility of e -Services in a Cooperative Multi-platform Environment. In *VLDB-TES'01*, Rome, Italy, 2001. Springer.
12. M. P. Papazoglou and D. Georgakopoulos. Special issue on service oriented computing. *Commun. ACM*, 46(10):24–28, 2003.
13. D.M. Yellin and R.E. Storm. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, March 1997.