Compatibility and replaceability analysis for timed web service protocols

Boualem Benatallah
boualem@cse.unsw.edu.au
CSE, UNSW
Sydney NSW 2052, Autralia

Fabio Casati
casati@hpl.hp.com
Hewlett-Packard Laboratories
Palo Alto, CA, 94304 USA

Julien Ponge, Farouk Toumani
{ponge,ftoumani}@isima.fr
Laboratoire LIMOS - CNRS UMR 6158
ISIMA - Campus des Cézeaux
63173 Aubière cedex, France

Abstract

Web services technology is emerging as the main pillar of service-oriented architectures (SOAs). This technology facilitates application integration by enabling programmatic access to applications through standard, XML-based languages and protocols. While much progress has been made toward providing basic interoperability among applications, there are still many needs and unexploited opportunities in this area. In particular, services in SOAs require richer description models than object or component interfaces. This is due to the loose coupling inherent in SOAs and therefore to the fact that services are developed independently of clients. Hence, service descriptions need to include all the information needed by clients to understand if they can interact with a service and how. As agreed by many researchers and practitioners, service descriptions should include not only the service interface, but also the *business protocol* of the service, i.e., the specification of which message exchange sequences are supported by the service. This paper discusses the augmentation of business protocols with specifications of temporal abstractions (e.g., temporal constraints on when an operation must or can be invoked), focusing in particular on problems related to compatibility and replaceability analysis. It defines concepts and provides primitives for analyzing compatibility between the protocols of requesters and providers and for analyzing similarities (replaceability) between the protocols of two providers. We describe these notions both informally and formally, and provide algorithms to check temporal compatibility and replaceability.

1 Introduction

Web services are increasingly gaining acceptance as a framework for facilitating application-to-application interactions within and across enterprises. Application interoperability has been and still is a difficult issue due to difficulties created by heterogeneous and autonomous systems. Web services provide abstractions and technologies for exposing enterprise applications as services and make them accessible programmatically through standardized interfaces. Indeed, the main benefit they bring to application integration is that of standardization, in terms of description languages, coordination, and interaction protocols [1, 16]. Standardization at interface definition language (WSDL) and transport protocol (SOAP) has enabled basic interoperability at messaging layer. Indeed, developers are beginning to use SOAP and WSDL to integrate enterprise applications [18].

While much progress has been made toward providing basic interoperability, there is still a lot to be done to simplify service development and interaction. In particular, an important aspect of Web services that affects interoperability is that services are loosely-coupled, that is, are not developed only to interact with specific clients but are meant to serve the needs of many different clients, possibly developed by different teams or even different companies. Hence, developers of client applications need to be aware of all functional and non-functional aspects of a service to be able to understand if they can/need interoperate with a service and how to develop clients that can interact correctly with the service. For this reason, service descriptions are richer than "just" descriptions of interfaces as in conventional middleware. Specifically, it is commonly accepted that a service description should include not only the interface, but also the *business protocol* supported by the service, i.e., the specification of possible message exchange sequences (conversations) that are supported by the service [5].

Tools supporting service development today are mainly concerned with interoperability at lower levels of service stack (e.g. mappings from WSDL to java and vice versa, making two SOAP-based systems talk to each other). There is little support for high level modeling and analysis of abstractions at higher level of services stack, and in particular there is little support for protocol modeling and management. We believe that indeed protocol modeling and management will be key in supporting Web service development and interaction, and that developing formal models and a protocol algebra will have a positive impact similar to the one that the relational model and the relational algebra had in database technology. The importance of formal analysis of service protocols in terms of automated support to services interoperability at the business protocol level has been discussed in some recent papers (e.g., [9, 10, 15, 11]).

When developing our framework for service protocols modeling, analysis, and management [5, 9], we identified the need for representing temporal abstractions in protocol descriptions. In particular, our analysis of the characteristics and requirements of service protocols in terms of description languages, we found that, in addition to message choreography constraints, protocol specification languages need to cater for time-sensitive conversations (i.e., conversations that are characterized by temporal constraints on when an operation must or can be invoked). For example, a protocol may specify that a purchase order message is accepted only if it is received within 24 hours after a quotation has been received. In this paper, we discuss the augmentation of business protocols with specifications of temporal abstractions (called timed protocols). We show that temporal abstractions are a crucial aspect of protocol modeling and we show how these can be modeled. Then we provide mechanisms for analyzing timed protocol specifications, and specifically for identifying if and under what conditions two services, characterized by certain timed protocols, can interact. We also provide abstractions and operators to verify whether a service, characterized by a certain protocol, can be used to substitute another service. We motivate why this kind of analysis is important in simplifying service development and binding, informally introduce the related problems, and then formally define concepts and algorithms.

This paper is structured as follows. We start by presenting the model we propose to represent temporal abstractions of business protocols (Section 2). We then motivate and discuss the need for protocol compatibility and replaceability analysis (Section 3). Section 4 defines operators that enable characterizing compatibility and replaceability classes for timed protocols while Section 5 describes the corresponding algorithms. Finally, in Section 6, we review related work and provide some concluding remarks. For space reasons, proofs are omitted. They can be found instead in [8].

2 Timed business protocols

This section first presents informally the *timed business protocols* which extend the *business protocols* to take into account temporal abstractions. We then give a formal definition.

In our approach, business protocols are modeled as deterministic finite state machines, where the states represent the different phases in which a service may go through during its interaction with a requester. Transitions are triggered by messages sent by the requester 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 choice of determinism comes from the observation that allowing multiple possible target states after a given transition will make protocols ambiguous in the sense that a service can move to a state that cannot be predicted by its clients. See [6, 1] for more details on the use of state machines for modeling protocols and for a discussion of why they are a well suited formalism.

Example 1 *As an example, Figure 1 shows a graphical representation of a protocol* P *that describes the external behavior of an order management service that allows users to buy some kinds of goods. Each transition is labeled with a message name followed by the message polarity, that is, whether the message is incoming (plus sign) or outgoing (minus sign) [20]. In this paper, we use the notation $m(+)$ (respectively, $m(-)$) to denote that m is an input (respectively, output) message. For instance, the protocol* P *specifies that the order management 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* $\mathsf{login}(+)$*), etc.*

For simplicity, and w.l.o.g., we do not model message replies in the examples (e.g., at Figure 1 there is no message $\mathsf{quoteReply}(-)$ that may be used by a service to reply to a $\mathsf{quoteRequest}(+)$ message). In fact, we can assume that transitions can be labeled with operations (i.e., either messages or pairs of request/reply of messages).

2.1 Extending business protocols with temporal abstractions

In our previous work on protocol modeling [5, 7], we identified that catering for temporal abstractions in protocol descriptions is an important requirement. In particular, our analysis of the characteristics and requirements of service protocols in terms of description languages, we found that, although most state transitions occur due to explicit operation invocations, there are cases in which transitions occur without an explicit invocation by requesters. We refer to these transitions as *implicit transitions*. The large majority of implicit transitions are due to timing issues (deadline expirations). For example, many services allow requesters to reserve a resource or to perform certain actions (invoke certain operations) only within a time window, after which these operations cannot be performed any more. There are countless examples of this behavior, such as airline companies allowing users to hold unticketed reservations only for a certain time period, or goods sellers allowing returns within a specified number of days. To model this important class of behaviors, we introduce the notion of *timed transitions*. A timed transition occurs automatically after a time interval is elapsed since the transition is *enabled* (i.e., the conversation state is

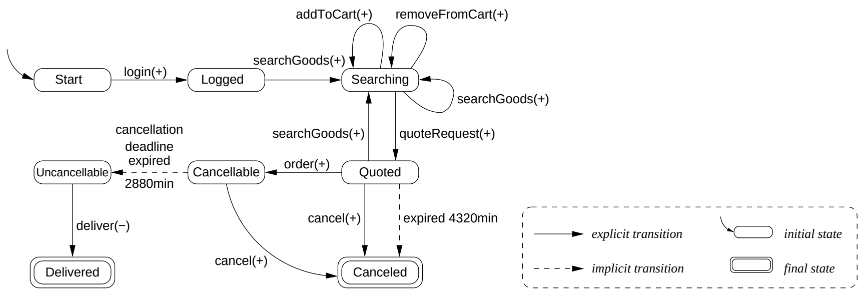

Figure 1: A sample timed business protocol P.

the transition's source state), or as a certain date and time is reached.

Example 2 *Consider again the protocol* P *of Figure 1. This protocol specifies that a user interacting with a service supporting this protocol needs first to invoke the* login *operation and then the* searchGoods *operation. Then, at the state* Searching*, users can add/remove goods from their shopping cart (operations* addToCart *and* removeFromCart *respectively) or search for other goods (operation* searchGoods*). After that, the user can ask for a price quotation by invoking the operation* quoteRequest *that leads the service to the state* Quoted*. The quotation is valid only for 3 days (equal to 4320 minutes), a time interval within which the user can order the selected goods (operation* order*). After this period of time, the conversation moves to the* canceled *state, denoting that the server has canceled the order (implicit transition* expired *with a temporal constraint 4320min). The* canceled *state is a final state, and the operation* order *cannot be invoked from that state. Note that the implicit transition* expired *imposes time constraints on all transitions that can be fired from state* Quoted *(i.e., the operations* order*,* searchGoods *and* cancel*), as once it fires it leads the conversation to a state from which those operations are not allowed. An analogous reasoning can be applied to transition* Cancellation deadline expired*.*

We use the term *timed business protocol* (or timed protocol for short) to denote a business protocol whose definition contains timed transitions. The next two subsections define formally timed protocols and give their associated trace semantics.

2.2 Formalization

We present here an extended definition of protocols that allows to cater for timing constraints.

Definition 1 ***(Timed business protocol)***

A timed 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, with $s_0 \in \mathcal{S}$ 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.*
- *$\mathtt{M} = \mathtt{M_e} \bigcup \mathtt{M_i}$ (such that $\mathtt{M_e} \bigcap \mathtt{M_i} = \emptyset$) is a finite set of messages where $\mathtt{M_e}$ is a finite set of explicit messages while $\mathtt{M_i}$ is a finite set of implicit messages. For each message $m \in \mathtt{M_e}$, 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 $\mathcal{P}$.*
- *$\mathcal{R} \subseteq \mathcal{S}^2 \times \mathtt{M}$ is a finite set of transitions. Each transition (s, s', m) identifies a source state s, a target state s' and either an explicit or an implicit message m. In this case, we say that the message m is enabled from a state s. In the sequel, we use $\mathcal{R}(s, s', m)$ to denote the fact that $(s, s', m) \in \mathcal{R}$ and*

we denote by $output_e(s)$ the set of explicit messages that are enabled from a state s.

- *If $\mathcal{R}(s, s', m)$ and $m \in \mathtt{M_i}$, then we define a function $Time(s, m)$ that returns a rational number $t \in \mathbb{Q}^{\geq 0}$ that specifies the timing constraints of m.*

We assume that transitions are instantaneous while time elapses in the states. Note that time constraints on a given transition $\mathcal{R}(s, s', m)$ are expressed relatively to the entering date into the state s. For example, the previous time constraint on the transition expire of the protocol P of figure 1 is specified as follows: $Time(\mathsf{Quoted}, \mathsf{expire}) = 4320min$.

We consider timed business protocols as deterministic systems, i.e., they verify the following two conditions: (i) a protocol has only one initial state, and (ii) for every state s and for every explicit message m of a given protocol P, there is at most one state s' of P such that the transition (s, s', m) holds in P. Also we assume that there is at most only one implicit transition that can be enabled at a state s as, even if there are many, at most only one can be fired thus preempting the others (we do not deal in this paper with parallel protocols). Finally, the two conditions given below enable to verify whether a given protocol is correct or not, i.e, whether an instantiation of the protocol is guaranteed to be executable: (i) A protocol must be *deadlock free* (i.e, there does not exist states that cannot reach a final state). This may be achieved by removing from protocol definitions any state that does not belong to a complete path starting from the initial state and ending in a final state, and (ii) A protocol must not contain cycles of implicit transitions. In the remaining of this paper, we assume that timed protocols are correct.

2.3 Timed protocol semantics

A timed business protocol defines two kinds of constraints on the external visible behavior of a given service:

(i) the conversations, expressed in term of sequences of messages exchanges, that a service supports. For example, the sequence of message exchange $\mathsf{login}(+)\cdot\mathsf{searchGoods}(+)\cdot\mathsf{addToCart}(+)$ is allowed by (compliant with) the protocol P of figure 1 while the sequence $\mathsf{login}(+)\cdot\mathsf{addToCart}(+)\cdot\mathsf{searchGoods}(+)$ is not compliant with P.

(ii) timing constraints that specify when a given messages is enabled to occur inside a conversation. For example, taking into account time constraints, the sequence $\mathsf{login}(+)\cdot\mathsf{searchGoods}(+)\cdot\mathsf{addToCart}(+)\cdot\mathsf{quoteRequest}(+)\cdot\mathsf{order}(+)$ will be compliant with the protocol P of figure 1 only if the message order is received before 4320 minutes after the message quotation has been received.

Condition (i) above can be characterized using the so-called *linear time* process semantics (see [14] for details on the various process model semantics). In such a semantics, a process is completely determined from the set of its (partial) observable runs (or traces). Following this approach, the behavior of a protocol will be characterized in terms of all its observable *traces*. For example, the sequence of messages $\mathsf{login}(+)\cdot\mathsf{searchGood}(+)\cdot\mathsf{addToCart}(+)$ is an execution trace (or simply, a trace) of the protocol P. We are particularly interested in the complete traces (i.e., traces that start from an initial state and end at a final state). For example, the sequence $\mathsf{login}(+)\cdot\mathsf{searchGoods}(+)\cdot\mathsf{addToCart}(+)\cdot\mathsf{quoteRequest}(+)\cdot\mathsf{cancel}(+)$ is a complete trace of the protocol P of Figure 1.

To characterize condition (ii), we need to extend the notion of trace to cater for timing constraints. Consider an execution of a service S that supports a protocol P. If m is an input message in P, we use the expression $(\mathsf{m}(+), t)$ to denote the reception of the message m at an instant t. t represents the elapsed time since the beginning of the execution of S (i.e., the beginning of the conversation in which m is involved). The pair $(\mathsf{m}(+), t)$ is called an input event. Similarly, a pair $(\mathsf{m}(-), t)$ is called an output event and represents the fact that the service issues a message login at an instant t. An event (m, t) is called implicit if m is an implicit message. A *timed trace*[1] is then defined as a sequence of input and output events $(\mathsf{a}_0, t_0), \ldots, (\mathsf{a}_\mathsf{n}, t_n)$ where the occurrence of times increase monotonically, i.e., $t_0 \leq t_1 \leq \ldots \leq t_n$. As an example, the sequence of events $(\mathsf{login}(+), 0) \cdot (\mathsf{searchGoods}(+), 1) \cdot (\mathsf{addToCart}(+), 3) \cdot (\mathsf{quoteRequest}(+), 7) \cdot (\mathsf{cancel}(+), 120)$ is a timed trace.

We introduce below the notion of executions of a timed protocols that will be useful to formally define timed traces.

Definition 2 ***(Executions of timed protocols)***
Let $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ be a timed protocol. An execution $\sigma = s_0.(\mathsf{a}_0, t_0).s_1 \ldots s_{n-1}.(\mathsf{a}_{n-1}, t_{n-1}).s_n$ of $\mathcal{P}$ is an alternating sequence of states and events of $\mathcal{P}$, starting and ending with a state, such that $\forall j \in [0, n\text{-}1]$, we have $\mathcal{R}(s_j, s_{j+1}, a_j)$ and:

- *if $a_j \in \mathtt{M}_\mathsf{e}$ and there exists a state s' such that $\mathcal{R}(s_j, s', m)$ with $m \in \mathtt{M}_\mathtt{i}$, then $t_j\text{-}t_{j-1} < Time(s_j, m)$, or*
- *if $a_j \in \mathtt{M}_\mathtt{i}$, then $t_j\text{-}t_{j-1} = Time(s_j, a_j)$*

If s_0 is the initial state and s_n a final state of $\mathcal{P}$, then σ is called a complete execution of $\mathcal{P}$.

We define now the notion of timed traces.

Definition 3 ***(Compliance of timed traces with a protocol)***
Let $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ be a timed protocol. A sequence of events $\tau = (\mathsf{a}_0, t_0), \ldots (\mathsf{a}_\mathsf{n}, t_n)$ is a timed trace of $\mathcal{P}$ iff there exists a set of states $s_0, \ldots, s_n$ in $\mathcal{S}$ such that $\sigma_\tau = s_0.(\mathsf{a}_0, t_0).s_1 \ldots, s_{n-1}.(\mathsf{a}_{n-1}, t_{n-1}).s_n$ is an execution of $\mathcal{P}$.

The trace τ is compliant with $\mathcal{P}$ if σ_τ is a complete execution of $\mathcal{P}$.

An ***observable*** *trace of $\mathcal{P}$ is a trace obtained from a complete timed trace of $\mathcal{P}$ by removing the implicit events.*

Note that we are only interested by the *observable* timed traces of a protocol (i.e., traces containing only explicit messages). Implicit messages act as silent actions which are not externally observables. In the sequel, and by abuse of language, we use the term timed traces (or simply traces) to refer to complete timed **observable** traces of a protocol. Informally, a timed trace is compliant with a given protocol if the ordering of the messages in the trace is compliant with the ordering prescribed by the protocol and each message that appear in an event of the trace is received/issued at an instant that satisfies the timing constraints of that protocol.

Example 3 *Consider an execution of a service* S *that supports the protocol* P *depicted at Figure 1. The timed trace* $(\mathsf{login}(+), 0) \cdot (\mathsf{searchGoods}(+), 1) \cdot (\mathsf{addToCart}(+), 3) \cdot (\mathsf{quoteRequest}(+), 7) \cdot (\mathsf{cancel}(+), 120)$ *is a timed trace which is compliant with the protocol* P*. However, the timed trace* $(\mathsf{login}(+), 0) \cdot (\mathsf{searchGoods}(+), 1) \cdot (\mathsf{addToCart}(+), 3) \cdot (\mathsf{quoteRequest}(+), 7) \cdot (\mathsf{cancel}(+), 262807)$ *is not compliant with the protocol* P *because the operation* cancel *is invoked 4380 minutes after the operation* quoteRequest *(i.e., after the service entered the state* Quoted*).*

Since we deal with deterministic systems, the semantics of a timed protocol $\mathcal{P}$ can be characterized by the set of all the timed traces which are compliant with that protocol. In the following, given a protocol $\mathcal{P}$, we note by $Tr(\mathcal{P})$ the set of all (observable) timed traces of $\mathcal{P}$. Moreover, timed protocols can be compared with respect to their semantics as given below.

Definition 4 ***(Timed subsumption and timed equivalence)***

[1]The notion of timed trace is inspired from [3] where it is called *timed word.*

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed protocols. Then, $\mathcal{P}_1$ subsumes $\mathcal{P}_2$, noted $\mathcal{P}_2 \lesssim_T \mathcal{P}_1$, iff $Tr(\mathcal{P}_2) \subseteq Tr(\mathcal{P}_1)$.

$\mathcal{P}_1$ and $\mathcal{P}_2$ are said to be equivalent, noted $\mathcal{P}_1 \cong_T \mathcal{P}_2$, iff $Tr(\mathcal{P}_2) = Tr(\mathcal{P}_1)$.

2.4 Timed protocol interaction

Interactions between two given timed protocols can also be characterized in terms of complete timed traces as illustrated by the example below.

Example 4 *Consider protocol* P *depicted on Figure 1 and its reversed protocol* P' *obtained from* P *by reversing the polarity of the messages (i.e., input messages becomes outputs and vice versa).* P' *can interact correctly with* P *in the sense that, considering a given interaction between these two protocols, whenever* P' *sends a message at an instant t, the protocol* P *could receive it and vice versa. For example, protocol* P' *supports the following complete timed trace:* $(\mathsf{login}(-), 0) \cdot (\mathsf{searchGoods}(-), 1) \cdot (\mathsf{addToCart}(-), 2) \cdot (\mathsf{quoteRequest}(-), 3) \cdot (\mathsf{cancel}(-), 4)$. *In this trace,* cancel *is the only operation whose temporal availability is restricted since both* P *and* P' *have an implicit transition that is fired 4320 minutes after having entered the* Quoted *state. In the previous trace the* cancel *message is sent by* P' *only 1 minute after the quotation has been performed, hence the message is legal.*

Indeed, from the previous example, we can observe that when P' interacts with P following a given timed trace τ, protocol P follows exactly a similar trace but with inverse polarity of messages. If P' is executed according to the trace given above, and if it is interacting correctly with P, so P will necessarily follow the execution trace: $(\mathsf{login}(+), 0) \cdot (\mathsf{searchGoods}(+), 1) \cdot (\mathsf{addToCart}(+), 2) \cdot (\mathsf{quoteRequest}(+), 3) \cdot (\mathsf{cancel}(+), 4)$. In this case, we call the path $(\mathsf{login}, 0) \cdot (\mathsf{searchGoods}, 1) \cdot (\mathsf{addToCart}, 2) \cdot (\mathsf{quoteRequest}, 3) \cdot (\mathsf{cancel}, 4)$ a *timed interaction trace* of P and P'. Note that, polarity of messages that appear in interaction traces is not defined as in such traces each input message m of one protocol coincides with an output message m of the other protocol.

Let τ be a timed trace of a protocol $\mathcal{P}$. In the sequel, we denote by $\overline{\tau}$ the inverse trace of τ (i.e., the trace obtained from τ by inverting polarity of messages) and by $Unp(\tau)$ the trace obtained from τ by removing polarity of messages. For example, if $\tau = (\mathsf{m}_1(+), t_1) \cdot (\mathsf{m}_2(-), t_2)$ then $\overline{\tau} = (\mathsf{m}_1(-), t_1) \cdot (\mathsf{m}_2(+), t_2)$ while $Unp(\tau) = (\mathsf{m}_1, t_1) \cdot (\mathsf{m}_2, t_2)$.

Definition 5 ***(Timed interaction traces)***

Let $\mathcal{P}$ and $\mathcal{P}'$ be a timed protocol and let $\tau = (\mathsf{a}_0, t_0), \ldots (\mathsf{a}_n, t_n)$ be a sequence of events in which the polarity of messages is not defined. Then τ is a timed interaction trace of $\mathcal{P}$ and $\mathcal{P}'$ iff there exists two timed traces τ_1 and τ_2 such that: (i) $\tau_1 \in Tr(\mathcal{P})$ and $\tau_2 \in Tr(\mathcal{P}')$, and (ii) $\tau_1 = \overline{\tau_2}$, and (iii) $\tau = Unp(\tau_1) = Unp(\tau_2)$.

Continuing with the example 4, $(\mathsf{login}, 0) \cdot (\mathsf{searchGoods}, 1) \cdot (\mathsf{addToCart}, 2) \cdot (\mathsf{quoteRequest}, 3) \cdot (\mathsf{cancel}, 4)$ is a timed interaction trace of protocols P and P' since the timed trace $(\mathsf{login}(-), 0) \cdot (\mathsf{searchGoods}(-), 1) \cdot (\mathsf{addToCart}(-), 2) \cdot (\mathsf{quoteRequest}(-), 3) \cdot (\mathsf{cancel}(-), 4)$ is compliant with P' while its reversed trace is compliant with P.

3 Compatibility and replaceability analysis

This section first discusses the problem and motivates the need for protocol analysis (and specifically compatibility and replaceability analysis). Then, it focuses on the specific issues related to timed business protocols.

In the introduction we have presented protocols as an important part of a service description as it helps developers how to write clients that interact with a service. There are, however, many other benefits that protocol modeling can provide, especially in terms of automated support to service development and binding. Specifically, we argue that there is a need for formal protocol analysis techniques and for a protocol algebra that allows users to manipulate and compare protocol definitions. More specifically, once services are endowed with protocol specifications, protocol management operators can be identified to perform the following type of analysis:

- *compatibility analysis*: this refers to checking if the protocol of a requester and of a provider are compatible, that is, if conversations can take place between the services.

- *replaceability analysis*: this refers to checking if a service provider R can replace another service provider S from a protocol standpoint, that is, if R can support the same conversation that S supports.

- *compliance analysis*: while the previous two analysis focus on comparing protocols of two services, compliance focuses on verifying whether the implementation of a service actually supports the declared protocol.

- *correctness analysis*: this refers to verifying the protocol definition to ensure that it has certain properties, e.g., that there is always a path in the protocol that allows conversations to end in a final state.

A proper discussion of all the above aspects would be way too lengthy to fit in a single paper. Hence, in the following we focus on compatibility and replaceability (we refer the reader to [4] for discussion on protocol compliance). These are very important properties of pairs of protocols, and their automated verification is very beneficial. Specifically, compatibility analysis is useful at both service development time and at binding time. It supports developments of clients since it helps developers verify if the service they have created to interact with a certain provider can actually hold a conversation with that provider. It supports binding since when a client is searching for suitable services, compatibility is one of the factors to be considered as there is no use in selecting a service with which the client is not compatible. Replaceability analysis becomes very handy in many important situations: for example, when developers create a new version of a service, they need to verify if the new service can hold all conversations of the previous version or, if not, they need to be aware of which conversations are now disallowed. Furthermore, as standardization in Web services continues to take place, it is likely that consortia will provide standard specifications for Web services protocols in specific business domain. RosettaNet, for example, defines standard business protocols for services in the IT supply chain space. Hence, service developers that aim at supporting a certain standard specification must be able to verify if their protocol can hold the same conversation as the one specified by the standard.

For both compatibility and replaceability, we have defined both *classes* to identify different levels of compatibility and replaceability, as well as *operators* that can be applied to protocol definition to asses the level of compatibility and replaceability. We introduce two compatibility classes below:

- *Partial compatibility* (or simply, compatibility): A protocol P_1 is partially compatible with another protocol P_2 if there are some executions of P_1 that can interoperate with P_2, i.e., if there is at least one possible conversation that can take place among two services supporting these protocols

- *Full compatibility*: a protocol P_1 is fully compatible with another protocol P_2 if all the executions of P_1 can interoperate with P_2, i.e., any conversation that can be generated by P_1 is understood by P_2.

The following are replaceability classes.

- *Protocol equivalence w.r.t. replaceability*: two business protocols P_1 and P_2

are equivalently replaceable if they can be interchangeably used in any context and the change is transparent to clients.

- *Protocol subsumption w.r.t. replaceability*: a protocol P_2 is subsumed by another protocol P_1 w.r.t. replaceability if P_1 supports at least all the conversations that P_2 supports. In this case, protocol P_1 can be transparently used instead of P_2 but the opposite is not necessarily true.

- *Protocol replaceability with respect to a client protocol*: A protocol P_1 can replace another protocol P_2 with respect to a client protocol Pc if P_1 behaves as P_2 when interacting with a specific client protocol Pc.

- *Protocol replaceability with respect to an interaction role*: Let P_R be a business protocol. A protocol P_1 can replace another protocol P_2 with respect to a role P_R if P_1 behaves as P_2 when P_2 behaves as P_R. This replaceability class allows to identify executions of a protocol P_2 that can be replaced by protocol P_1 even when P_1 and P_2 are not comparable with respect to any of the previous replaceability classes.

- *Partial protocol replaceability*: for all of the above classes, we can distinguish between full and partial replaceability. Full replaceability is as defined above. Partial replaceability is when there is replaceability but only for some conversations and not others. For example, we have partial replaceability with respect to a client protocol when protocol P_1 can replace another protocol P_2 in at least some of the conversations that can occur with Pc.

In earlier work [9], we have identified operators to verify the compatibility or replaceability classes to which a pair of non-timed protocols belongs, as well as operators to extract the part of the protocols that are or are not compatible and replaceable. We have already stressed the importance of time modeling in protocols, which has led to the definition of the protocol model informally presented in the previous section and formally described later. Correspondingly, there is the need for revisiting and extending the concepts and the formalizations defined earlier for ordinary protocols to make them applicable to timed protocols, as well as for revisiting earlier formalizations and analysis to cater with this extended protocol model. This is far from easy, as the introduction of time aspects adds significant complexity to the problem. We next discuss the novel opportunities and needs that timed protocols bring in this regard, first informally by means of example and then formally.

3.1 Compatibility in timed protocols

We present here several examples related to protocol compatibility analysis, starting from a simple case to more complex ones. We describe several examples as there are several different aspects that need to be considered in timed protocol analysis. These examples will help us motivate the need for additional compatibility classes and operators, formally described in the next section. The same will be done later for replaceability.

We present below a simple example to illustrate incompatibility between two protocols.

Example 5 *Consider a protocol* P' *that supports the following timed trace:* $(\mathsf{login}(-), 0) \cdot (\mathsf{searchGoods}(-), 1) \cdot (\mathsf{addToCart}(-), 2) \cdot (\mathsf{quoteRequest}(-), 3) \cdot (\mathsf{cancel}(-), 2890)$. *During such an execution,* P' *cannot interact correctly with the protocol* P *of Figure 1. Indeed,* P' *will fire the operation* cancel *2890 minutes after the quotation has been performed, which is more than the 2890 minutes allowed by* P *(i.e.,* P *has already moved to the* $\mathsf{Canceled}$ *state).*

The previous case were simple to check because it was sufficient to compare pairs of

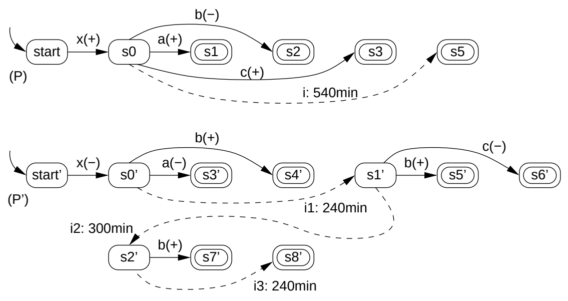

Figure 2: Two compatible timed protocols.

states locally. The following example illustrates a more complex case.

Example 6 *Consider the protocols* P *and* P′ *depicted on Figure 2. Unlike the previous examples, the two protocols have very different shapes. For instance, we can observe that after the execution of the operation* x *the protocols* P *and* P′ *move, respectively, to the states* s_0 *and* s'_0*. These states do not offer the same operations (at least if we consider the operations that are defined explicitly at these two states). The state* s_0 *provides the operations* a, b *and* c *while the state* s'_0 *only provides the operations* a *and* b*. Consequently, focusing compatibility checking only on these two states is not enough. Indeed the operation* c *for example may be available for a client interacting with this service after 240 minutes. This is due to the presence of an implicit transition* i_1 *that automatically leads a service to the state* s'_1 *from which* c *can be fired.*

Consequently, to be able to check compatibility between protocols there is a need for a mechanism that makes explicit all the operations that are available (i.e., can be fired) at a given state as well as their associated timing constraints.

Example 7 *Continuing with the example, to check if protocol* P *and* P′ *are compatible we need also to consider all the states that are automatically (implicitly) reachable from a given state. In our case, checking if* s_0 *and* s'_0 *are compatible implies that we also consider* s'_1 *and* s'_2 *since they can be reached from* s'_0 *through* i_1 *and* i_2*. More precisely, we need to make explicit all the operations, and their associated timing constraints, that are available at these states. For example, as given below, looking to the implicit transitions, we can derive the temporal availabilities of the operations at the states* s'_0 *and* s_0*:*

$$(\mathsf{P})\ \begin{cases} \mathsf{a}: [0min, 540min] \\ \mathsf{b}: [0min, 540min] \\ \mathsf{c}: [0min, 540min] \end{cases}$$
$$(\mathsf{P}')\ \begin{cases} \mathsf{a}: [0min, 240min] \\ \mathsf{b}: [0min, 780min] \\ \mathsf{c}: [240min, 540min] \end{cases}$$

Operation a *will be performed by* P′ *during a temporal window where* P *is ready to accept the message fired by* P′*. The same is true for* c*. The case of* b *is different since it is* P *that sends the message. The temporal window defined by* P′ *for receiving the related message is wider than the one used for* P *to send it, thus* P′ *is ready to receive a message* b *fired by* P*. We see that conversations can take place between* P *and* P′ *as the messages can be exchanged during the allowed temporal slices defined by each protocol. However the compatibility between* s_0 *and* s'_0 *is not obvious since several other states have to be taken into account to get to the conclusion that a compatibility is effectively possible. However, note that compatibility is dependent on timing. In fact, not all conversations that can be generated by the client (*P*) can be supported by the provider (*P′*). If the client implementation is such that the client sends a message* c *right away after a message* x*, then it will cause the provider to respond with a fault message.*

Finally, the following example shows that implicit transitions can also influence the identification of final states and this naturally impacts compatibility analysis.

Example 8 *Let's consider the 3 protocols* P*,* P′ *and* P″ *depicted on Figure 3. We can observe that when interacting with* P′ *or* P″*, the protocol* P *will reach its final state after executing the operations* a *and* b *while* P′ *and* P″ *both remain at intermediary states*

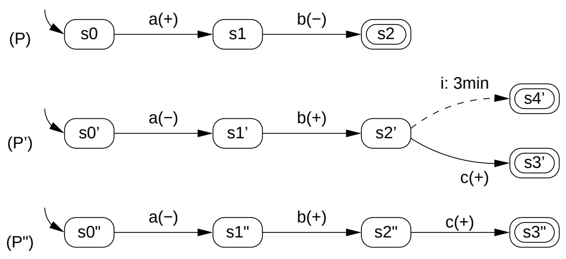

Figure 3: Another compatibility problem illustrated.

(respectively, the states s'_2 *and* s''_2*). Clearly, this is not a problem for* P' *as this protocol is able to automatically reach a final state from the state* s'_2*, and hence, its terminates correctly the conversation. Therefore, the interaction of* P *with* P' *is correct. However,* P *and* P'' *are not compatible since* P'' *remains in an intermediary state and will not be able to terminate correctly its execution (i.e., to reach a final state).*

The above discussion has emphasized the need for a new compatibility class, called *time-dependent compatibility.* Protocols P and P' have time-dependent compatibility if they are compatible only when they exchange messages following certain time constraints. Hence, time-dependent compatibility is a kind of partial compatibility. Note that an implementation of a client may be able to read the service provider's protocol and time its interaction so that messages are sent when allowed. The discussion of such "adaptive" implementations is outside the scope of this paper, since as mentioned here we limit to protocol analysis without discussing service implementation and compliance.

Correspondingly, the discussion has also emphasized the need for operators that identify these time constraints resulting from the joint compatibility analysis of the two protocols, as shown earlier.

3.2 Replaceability in timed protocols

We now turn our attention to the replaceability problem. We will provide below less examples as the previous section has already given an indication of the issues that can arise.

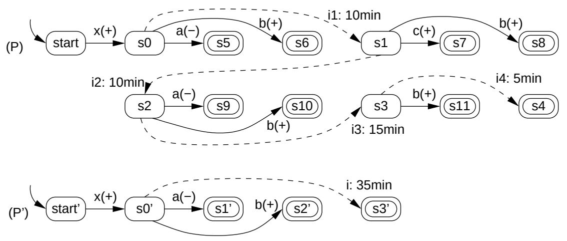

Figure 4: A protocol P that can replace P'.

Example 9 *Consider protocols* P *and* P' *depicted on Figure 4. Again, let's have a look at the states that are implicitly reachable from* s_0 *and* s'_0 *to compute the temporal availabilities of the operations:*

$$(\mathsf{P}) \left\{ \begin{array}{l} \mathsf{a}: [0min, 10min], [20min, 35min] \\ \mathsf{b}: [0min, 40min] \\ \mathsf{c}: [10min, 20min] \end{array} \right.$$

$$(\mathsf{P}') \left\{ \begin{array}{l} \mathsf{a}: [0min, 35min] \\ \mathsf{b}: [0min, 35min] \\ \mathsf{c}: \emptyset \end{array} \right.$$

Protocol P *can handle messages* x*,* a *and* b *from a client that is compatible with* P' *by looking at the temporal constraints. However* P *has an extra operation* c*. This operation being a message reception, it does not cause a problem. Indeed, a requester or* P' *does not know about* c *since* P' *does not support it. Thus, they will never attempt to fire it. Finally,* P *can well replace* P' *by looking at* s_0 *and* s'_0*.*

In summary, similarly to compatibility analysis, the introduction of time also emphasizes the need for a new replaceability class, called *time-dependent replaceability.* Protocol P can replace P' in a time-dependent manner if P can replace P' provided that clients send messages following certain time constraints. Again, this is a case of partial replaceability, and just like partiality, dependence on time is a "dimension" that applies to the other replaceability classes (that is, we can have time-dependent replaceability with respect to a client, time-dependent subsumption, and the like). In addition, users would benefit from operators that return the time constraints that define the time-dependent replaceability, to help them evaluate if these are acceptable or if the service should be modified to achieve replaceability independently of time issues.

4 Timed protocol management operators

In this section, we revisit the operators that we have defined for the business protocols [9], extending them to cater with temporal abstractions. Then we show how these operators can be used to characterize various timed compatibility and replaceability classes. Algorithms implementing the proposed operators are described in the next section.

4.1 Timed compatible composition

The operator *timed compatible composition* allows to characterize possible interactions between two timed business protocols, that of a provider and that of a requester. The resulting timed protocol describes all the timed interaction traces of the considered protocols and therefore characterizes the possible conversations that can take place between the requester and the provider. This operator, denoted as $\|^{\mathtt{TC}}$, takes as input two timed protocols and returns another one called a *(timed) compatible composition protocol.* Note that polarity of messages is not defined in *compatible composition protocols* since such protocols describe interaction traces.

Definition 6 ***(Timed compatible composition)***
Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed protocols. A protocol $\mathcal{P}$ is a timed compatible composition of $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P} = \mathcal{P}^1 \|^{\mathtt{TC}} \mathcal{P}^2$, iff $Tr(\mathcal{P})$ is the set of all the timed interaction traces of $\mathcal{P}_1$ and $\mathcal{P}_2$.

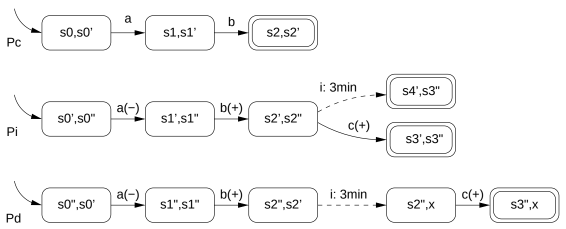

Figure 5: Protocols obtained by applying operators.

The previous definition gives the semantics of the timed compatible composition operator. Given two timed protocols, an algorithm that computes their compatible composition is described in the next section.

As an example, protocol P_c depicted on Figure 5 is obtained by $\mathsf{P} \|^{\mathtt{TC}} \mathsf{P}'$ where P and P' are the protocols depicted on Figure 3.

Let $\mathcal{P} = \mathcal{P}^1 \|^{\mathtt{TC}} \mathcal{P}^2$ be a timed compatible composition protocol. We note by $[\mathcal{P}]_{\mathcal{P}_1}$ the protocol obtained by defining the polarity of the messages in $\mathcal{P}$ similarly to those of $\mathcal{P}_1$ (i.e., for each message m in $\mathcal{P}$, we define $Polarity([\mathcal{P}]_{\mathcal{P}_1}, m) = Polarity(\mathcal{P}_1, m)$).

4.2 Timed intersection

The timed intersection operator allows the computation of the largest common part between two timed protocols. The operator, denoted as $\|^{\mathtt{TI}}$, takes as input two timed business protocols and returns a timed business protocol that describes the set of timed traces that are common to the two protocols. The resulting protocol is called a *timed intersection protocol.*

Definition 7 *(Timed intersection)*
Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed protocols. A protocol $\mathcal{P}$ is a timed intersection protocol of $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P} = \mathcal{P}^1 \|^{\mathtt{TI}} \mathcal{P}^2$, iff $Tr(\mathcal{P}) = Tr(\mathcal{P}_1) \cap Tr(\mathcal{P}_2)$.

The protocol P_i depicted on Figure 5 is obtained by $\mathsf{P}' \|^{\mathtt{TI}} \mathsf{P}''$ where P' and P'' are the protocols depicted on Figure 3.

4.3 Timed difference

While the timed intersection operator identifies common aspects between two protocols, the *timed difference operator*, denoted as $\|^{\mathtt{TD}}$, emphasizes their differences. This protocol takes as input two timed business protocols $\mathcal{P}_1$ and $\mathcal{P}_2$ and returns a timed business protocol (*the timed difference protocol*) whose purpose is to describe the set of all the timed traces compliant with $\mathcal{P}_1$ that are not compliant with $\mathcal{P}_2$.

Definition 8 ***(Timed difference)*** *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed business protocols. A timed protocol $\mathcal{P}$ is a timed difference protocol of $\mathcal{P}_1$ and $\mathcal{P}_2$, noted $\mathcal{P} = \mathcal{P}_1 \|^{\mathtt{TD}} \mathcal{P}_2$, iff $Tr(\mathcal{P}) = Tr(\mathcal{P}_1) \setminus Tr(\mathcal{P}_2)$*

The protocol P_d depicted on Figure 5 is obtained by $\mathsf{P}'' \|^{\mathtt{TD}} \mathsf{P}'$ where P' and P'' are the protocols depicted on Figure 3.

4.4 Characterizing compatibility and replaceability classes

We recall that a protocol $\mathcal{P}_1$ is partially compatible with a protocol $\mathcal{P}_2$ if there is at least one possible conversation that can take place among two services supporting these protocols, while protocol $\mathcal{P}_1$ is fully compatible with $\mathcal{P}_2$ if all the executions of $\mathcal{P}_1$ can interoperate with $\mathcal{P}_2$. We give below a formal definition of these classes.

Definition 9 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed business protocols.*

- *$\mathcal{P}_1$ is partially timed compatible with $\mathcal{P}_2$, noted PT-Compat($\mathcal{P}_1$,$\mathcal{P}_2$), iff there exists at least one timed interaction trace of $\mathcal{P}_1$ and $\mathcal{P}_2$.*

- *$\mathcal{P}_1$ is fully timed compatible with $\mathcal{P}_2$, noted FT-Compat($\mathcal{P}_1$,$\mathcal{P}_2$), iff $\forall \tau \in Tr(\mathcal{P}_1)$, then $Unp(\tau)$ is a timed interaction trace of $\mathcal{P}_1$ and $\mathcal{P}_2$.*

Based on the operators introduced above, the following lemma gives necessary and sufficient conditions to identify the compatibility level between two protocols.

Lemma 1 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed business protocols.*

1. *PT-Compat($\mathcal{P}_1$,$\mathcal{P}_2$) iff $\mathcal{P}_1 \|^{\mathtt{TC}} \mathcal{P}_2$ is not an empty protocol (i.e., the set of its final states is not empty).*

2. *FT-Compat($\mathcal{P}_1$,$\mathcal{P}_2$) iff $[\mathcal{P}_1 \|^{\mathtt{TC}} \mathcal{P}_2]_{\mathcal{P}_1} \cong_T \mathcal{P}_1$.*

4.5 Replaceability classes

We present first formal definitions of timed replaceability classes.

Definition 10 *Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two timed business protocols.*

- *$\mathcal{P}_1$ subsumes $\mathcal{P}_2$ w.r.t. replaceability, noted TSubs($\mathcal{P}_1$,$\mathcal{P}_2$), iff for every timed interaction trace $Unp(\tau)$ of $\mathcal{P}_2$ and any timed protocol $\mathcal{P}_c$ such that $\tau \in Tr(\mathcal{P}_2)$ then $\tau \in Tr(\mathcal{P}_1)$. Note that, in this case $\mathcal{P}_1$ can be transparently used instead of $\mathcal{P}_2$.*

- *$\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent w.r.t. replaceability, noted TEquiv($\mathcal{P}_1$,$\mathcal{P}_2$), iff Subs($\mathcal{P}_2$,$\mathcal{P}_1$) and Subs($\mathcal{P}_1$,$\mathcal{P}_2$).*

- *Protocol $\mathcal{P}_1$ can replace $\mathcal{P}_2$ with respect to a client protocol $\mathcal{P}_C$, denoted as $TRepl_{[\mathcal{P}_C]}(\mathcal{P}_1,\mathcal{P}_2)$, if for every timed interaction trace $Unp(\tau)$ of $\mathcal{P}_2$ and $\mathcal{P}_C$ such that $\tau \in Tr(\mathcal{P}_2)$, then $\tau \in Tr(\mathcal{P}_1)$.*

- *Let $\mathcal{P}_R$ be a protocol. Protocol $\mathcal{P}_1$ can replace $\mathcal{P}_2$ with respect to a role $\mathcal{P}_R$, denoted as $TReplRole_{[\mathcal{P}_R]}(\mathcal{P}_1,\mathcal{P}_2)$, if for every complete interaction trace $Unp(\tau)$ of $\mathcal{P}_2$ and any protocol $\mathcal{P}_C$ the following conditions hold: if $\tau \in Tr(\mathcal{P}_R)$ then $\tau \in Tr(\mathcal{P}_1)$. In this case, $\mathcal{P}_1$ can transparently replace $\mathcal{P}_2$ when $\mathcal{P}_2$ behaves as $\mathcal{P}_R$.*

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

Lemma 2 *Let $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_C$ and $\mathcal{P}_R$ be timed business protocols.*

1. *TSubs($\mathcal{P}_1$,$\mathcal{P}_2$) iff $\mathcal{P}_2 \lesssim_T \mathcal{P}_1$.*

2. *$TEquiv(\mathcal{P}_1, \mathcal{P}_2)$ iff $\mathcal{P}_1 \cong_T \mathcal{P}_2$.*

3. *$TRepl_{\mathcal{P}_C}(\mathcal{P}_1, \mathcal{P}_2)$ iff $\left[\mathcal{P}_C \parallel^{\texttt{TC}} \mathcal{P}_2\right]_{\mathcal{P}_2} \lesssim_T \mathcal{P}_1$ (or equivalently iff $\mathcal{P}_C \parallel^{\texttt{TC}} (\mathcal{P}_2 \parallel^{\texttt{TD}} \mathcal{P}_1)$ is an empty protocol).*

4. *$TReplRole_{[\mathcal{P}_R]}(\mathcal{P}_1, \mathcal{P}_2)$ iff $\mathcal{P}_R \parallel^{\texttt{TI}} \mathcal{P}_2 \lesssim_T \mathcal{P}_1$.*

5 Algorithms

The previous section, mainly lemma 1 and 2, stresses the need of operators to effectively handle timed protocol compatibility and replaceability analysis. We focus our attention now on the implementation of the operators. As illustrated in the previous examples (c.f., section 3), one difficulty for defining such formal tools comes from the fact that the presence of timed transitions makes many information about the availability of operations at a given state as well as their timing constraints implicit (i.e., hidden) in protocol definitions. In this section, we first describe an algorithm that allows to compute the temporal availability of operations at a given state. Our argument is that, with this knowledge at hands, it will be easier to devise the needed algorithms. We then present algorithms that implement timed compatible composition and timed difference as well as a timed subsumption test and give their computational complexity. Note that, a timed intersection algorithm can be easily derived from the compatible composition algorithm.

5.1 Expliciting state operations

The last sections have highlighted the fact that the set of operations available from a given state is not limited to the ones where it is the source state. Indeed, implicit transitions can enable the access to the operations of another state, thus adding new operations or disabling existing ones. For instance, in the protocol P of Figure 4, the operation a is available during the time intervals $[0min, 10min]$ and $[20min, 35min]$ after an execution of a service enters the state s_0.

This information is very useful in order to be able to analyse and compare timed protocols. This leads us to the definition of a data structure called *time-state space* that will be used to describe the informations about all the operations available at a given state. It should also be noted that it is of particular importance to keep trace of the target states in each case. For example, after entering the state s_0, we should record the fact that if the operation a of the protocol P is invoked during the interval $[0min, 10min]$ the the service will move to the state s_5 while if it is invoked during the interval $[20min, 35min]$ then the service will move to the state s_9 of the protocol P (*i.e.* we need to keep trace of the pairs $([0min, 10min], \mathsf{s}_5)$ and $([20min, 35min], \mathsf{s}_9)$).

We present here the formal definition of a time-state space structure.

Definition 11 ***(Time-state space)*** *Let $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ be a timed business protocol.*

- *A time-state tuple is a pair $([t_1, t_2], s)$ such that $t_1, t_2 \in \mathbb{Q}^{\geq 0}$, with $t_1 \leq t_2$, and $s \in \mathcal{S}$ is a state of P.*

- *A time-state space $\mathcal{T}$ is a set of time-state tuples that verify the following condition: $\forall([t_i, t_j], s), ([t_l, t_k], s) \in \mathcal{T}$ we have $t_j < t_l$ or $t_i > t_k$.*

The structure of a time-state space allows us to describe the operations available at a given state of a protocol, their associated timing constraints as well as the target states for each possible operation invocation. Continuing with the previous example, the time-space $\{([0min, 10min], \mathsf{s}_4), ([20min, 35min], \mathsf{s}_8)\}$ describes the temporal availability of the operation a at the state s_0 of protocol P.

It should be noted that, from the definition above, a time-state space does not allow overlapping time intervals between time-state tuples that have same target state. If such a case occurs, the tuples are merged. For example, the time-state tuples

Algorithm 1: The τ-*closure* algorithm.

```
Data: A protocol $\mathcal{P} = (\mathcal{S}, s_0, \mathcal{F}, \mathtt{M}, \mathcal{R})$ and a state $s \in \mathcal{S}$.
Result: $\tau$-closure$(s)$, T-Availability$(s, m_e)$, for each message $m_e \in output_e(s)$.
begin
1   upperBound := $+\infty$ ;
    foreach $s \in \mathcal{S}$ do
        if $\exists s' \in \mathcal{S} \,|\, \mathcal{R}(s, s', m_i) \wedge m_i \in \mathtt{M_i}$ then upperBound := Time$(s, m_i)$ ;
        foreach $m_e \in output_e(s)$ do
            initAvailability$(s, m_e)$ := $\{([0, upperBound], s')\}$ $(s' \,|\, \mathcal{R}(s, s', m_e))$ ;
2       upperBound := $+\infty$ ;
    $\tau$-closure$(s)$ := $\{s\}$, last := s, elapsed := 0, temp := $\emptyset$ ;
3   foreach $m_e \in \mathtt{M_e}$ do T-Availability$(s, m_e)$ := $\emptyset$ ;
    while temp $\neq \tau$-closure$(s)$ do
        temp := $\tau$-closure$(s)$ ;
        if $\exists s' \in \mathcal{S} \,|\, \mathcal{R}(last, s', m_i) \wedge m_i \in \mathtt{M_i}$ then
            last := $s'$ ;
            elapsed := elapsed + Time$(last, m_i)$ ;
            $\tau$-closure$(s)$ := $\tau$-closure$(s) \cup \{s'\}$ ;
            foreach $m_e \in output_e(s')$ do
                T-Availability$(s, m_e)$ := T-Availability$(s, m_e)$ $\cup_T$ Delay(initAvailability$(s', m_e)$, elapsed)
                ;
4
    return $\tau$-closure$(s)$, T-Availability$(s, m_e)$ $\forall m_e \in output_e(s)$ ;
end
```

$([0min, 10min], \mathsf{s}_4)$ and $([5min, 35min], \mathsf{s}_4)$ cannot appear together in a same time-state space but will be merged into one tuple: $([0min, 35min], \mathsf{s}_4)$.

When considering a timed business protocol, applying time-state space computations from the various states allows to reason on a protocol which contains the explicit messages with their effective temporal windows. This *explicited* won't be further detailed for space reasons but it is obvious that it is equivalent to a timed business protocol and the mapping from one to the other is straightforward.

We introduce some operations that facilitate the manipulation of time-state tuples and spaces.

Definition 12 *(Operations on time-state spaces)*

- *Time membership ($\in_T$)*

  We say that a time instant $t \in \mathbb{Q}^{\geq 0}$ belongs to a time-state space $\mathcal{T}$, noted $t \in_T \mathcal{T}$, iff there exists a time-state tuple $([t_i, t_j], s) \in \mathcal{T}$ such that $t \in ([t_i, t_j], s)$ (i.e., $t_i \leq t \leq t_j$). We also say that a temporal interval $[t_a, t_b]$ (with $t_a, t_b \in \mathbb{Q}^{\geq 0}$) belongs to a time-state space $\mathcal{T}$, noted $[t_a, t_b] \in_T \mathcal{T}$, iff there exists a time-state space tuple $([t_i, t_j], s) \in \mathcal{T}$ such that $t_i \leq t_a$ and $t_b \leq t_j$.

- *Delaying time-state spaces (Delay)*

  This operation introduces a delay in the time intervals of time-state space. Let $\mathcal{T}$ be a time-sate space and $t \in \mathbb{Q}^{\geq 0}$ a positive real number. We define the function $Delay(\mathcal{T}, t) = \{([t_i + t, t_j + t], s), \forall ([t_i, t_j], s) \in \mathcal{T}\}$.

- *Merging a set of tuples (Merge)*

  Let $S = \{([t_1, t_2'], s_1), \ldots, ([t_n, t_n'], s_n)\}$ be a set of tuples. We note by $Merge(S)$ the operation that allows to merge in S all the time-space tuples that have overlapping time intervals and similar target states. More precisely, the set $Merge(S)$ consists in the set S in which we recursively replace with the tuple $([Min(t_i, t_l), Max(t_j', t_k')], s)$ any pair of elements $([t_i, t_j'], s)$ and $([t_l, t_k'], s\prime)$ that verify: $s = s\prime$ and either $t_l \leq t_i \leq t_k'$ or $t_l \leq t_j' \leq t_k$.

- *Union ($\cup_T$)*

The union of two time-state spaces $\mathcal{T}_1$ and $\mathcal{T}_2$, noted $\mathcal{T}_1 \cup_T \mathcal{T}_2$ is a time-state space $\mathcal{T}$ obtained as follows: $\mathcal{T} = Merge(\mathcal{T}_1 \cup \mathcal{T}_2)$. Hence, time-state spaces are closed under the union operation.

Let us now consider the problem of computing temporal availabilities. The algorithm 1, called *τ-closure* algorithm, enables to achieve such a task. Given a protocol P and a state s in P, the algorithm first computes, for each state in P, the initial availabilities of messages as explicitly defined in P (lines 1 to 2). Note that, if no implicit transition is defined in a given state, then explicit messages enabled from this state will have $+\infty$ as an upper bound time constraint. Then, the remaining part of the algorithm (lines 3 to 4), consists in computing a transitive closure of the state s (the τ-closure) with respect to implicit transitions and incrementally updating the *T-Availability* values for the messages that can be fired from this state. Please note that, the algorithm uses a function $output_e$ which returns the set of explicit messages that can be fired from a given state.
Complexity analysis. Let n and m be, respectively, the number of states and the number of explicit transitions in the input timed protocol. The *τ-closure* algorithm runs in time $O(n * m)$.

5.2 Timed compatible composition and timed intersection

Algorithm 2 implements the timed compatible composition operator. The initial state of the resulting protocol is obtained by combining the initial states of the input protocols (line 2). The final states are obtained by combining the states (s_i, s_j) of the input protocols such that s_i (respectively, s_j) is either a final state or it can reach automatically a final state (line 4). Intermediary states of the resulting protocol are constructed by composing the input protocols messages of the input protocols that have same names but opposite polarities and overlapping time intervals ($foreach$ loop that starts at line 1). Also to ensure correctness of the resulting protocol, there is an additional pruning step which is not detailed here. This step consists in removing from the resulting protocol all the states that are not reachable from the initial state as well as the states that cannot lead to a final state.

Note that, an algorithm for implementing the timed intersection operator can be easily deduced from Algorithm 2 as the two operators differ only in how messages polarity is taken into account.
Complexity analysis. Let n and m be respectively the maximal number of states and the maximal number of explicit transitions in the input protocols. Algorithm 2 runs in time $O(n^2 m^2)$.

5.3 Timed difference

Algorithm 3 implements the timed difference operator. It is designed in the same spirit as the previous algorithm (i.e., it uses a layered approach to construct the resulting protocol). When exploring pairs of states from the input protocols $\mathcal{P}_1$ and $\mathcal{P}_2$, algorithm 3 identifies the outgoing messages of $\mathcal{P}_1$ that do not match with the outgoing messages of $\mathcal{P}_2$. Two cases can occur: (i) both input protocols have the same outgoing message with similar polarity (line 1), or (ii) $\mathcal{P}_1$ has an outgoing message that cannot be fired from the corresponding state of $\mathcal{P}_2$ (line 3). In case (i), an outgoing message, say m, will be added to the resulting timed difference protocol with a temporal availability that is enabled in $\mathcal{P}_1$ but not in $\mathcal{P}_2$. Such an availability is computed by intersecting the temporal availability of the message m in $\mathcal{P}_1$ with the complement of the temporal availability of m in $\mathcal{P}_2$. This operation is abbreviated in the algorithm as illustrated by the following example: $\{[0, +\infty[\} \cap \overline{\{[3, 4], [5, 6]\}} = \{[0, 3[,]4, 5[,]6, +\infty[\}$. In case (ii), the message from $\mathcal{P}_1$ is added with the same temporal availabilities as in $\mathcal{P}_1$. In this case, the

Algorithm 2: Timed compatible composition Algorithm.

```
Data: Two protocols $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ and $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$.
Result: A composition protocol expressed as a set of states $\mathcal{S}$, an initial state $s_0$, a set of final states $\mathcal{F}$ and
        for each $s \in \mathcal{S}$, a set $T\text{-}Availability(s, m_e)$ $\forall m_e \in output_e(s)$
begin
    $Candidates := \{(s_0^1, s_0^2)\}$, $\mathcal{S} := \emptyset$, $s_0 = (s_0^1, s_0^2)$, $\mathcal{F} := \emptyset$ ;
1   while $\exists(s^1, s^2) \in Candidates$ do
        $cur^1 := s^1$, $cur^2 := s^2$ ;
2       $\mathcal{S} := \mathcal{S} \cup \{(cur^1, cur^2)\}$ ;
3       foreach $m_e \in output_e(cur^1) \cap output_e(cur^2) \mid Polarity(\mathcal{P}_1, m_e) \neq Polarity(\mathcal{P}_2, m_e)$ do
            foreach $([t_i^1, t_j^1], s^1), ([t_l^2, t_k^2], s^2) \in$
            $T\text{-}Availability(cur1, m_e) \times T\text{-}Availability(cur2, m_e) \mid \max(t_i^1, t_l^2) \leq \min(t_j^1, t_k^2)$ do
                $t_{min} := \max(t_i^1, t_l^2)$, $t_{max} := \min(t_j^1, t_k^2)$ ;
                $T\text{-}Availability((cur^1, cur^2), m_e) :=$
                $T\text{-}Availability((cur^1, cur^2), m_e) \cup \{([t_{min}, t_{max}], S(s^1, s^2))\}$ ;
                $Candidates := Candidates \cup \{(s^1, s^2)\}$ ;
        $Candidates := Candidates \setminus \mathcal{S}$ ;
4       foreach $(s^1, s^2) \in \mathcal{S} \mid \tau\text{-}closure(s^1) \cap \mathcal{F}^1 \neq \emptyset \wedge \tau\text{-}closure(s^2) \cap \mathcal{F}^2 \neq \emptyset$ do
            $\mathcal{F} := \mathcal{F} \cup \{(s^1, s^2)\}$
    return $\mathcal{S}$, $s_0$, $\mathcal{F}$ and for each $s \in \mathcal{S}$, a set $T\text{-}Availability(s, m_e)$ $\forall m_e \in output_e(s)$ ;
end
```

Algorithm 3: A timed difference algorithm.

```
t
Data: Two protocols $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ and $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$.
Result: A difference protocol $\mathcal{P}$ expressed as a set of states $\mathcal{S}$, an initial state $s_0$, a set of final states $\mathcal{F}$ and
        for each $s \in \mathcal{S}$, a set $T\text{-}Availability(s, m_e)$ $\forall m_e \in output_e(s)$
begin
    $Candidates := \{(s_0^1, s_0^2)\}$, $\mathcal{S} := \emptyset$, $s_0 = (s_0^1, s_0^2)$, $\mathcal{F} := \emptyset$ ;
    while $\exists(s^1, s^2) \in Candidates$ do
        $cur^1 := s^1$, $cur^2 := s^2$ ;
        $\mathcal{S} := \mathcal{S} \cup \{(cur^1, cur^2)\}$ ;
        forall $m \in \mathtt{M}^1 \mid \mathcal{R}^1(cur^1, s_1^1, m)$ do
            Let $\{[t_i, t_j]\}$ $(i \leq j \in \mathbb{Q}^{\geq 0})$ such as $\exists([t_i, t_j], s_1^1) \in T\text{-}Availability(cur^1, m)$ ;
            $Polarity(\mathcal{P}, m) := Polarity(\mathcal{P}_1, m)$ ;
1           if $\mathcal{R}^2(cur^2, s_1^2, m) \wedge Polarity(\mathcal{P}_1, m) = Polarity(\mathcal{P}_2, m)$ then
                Let $\{[t_i', t_j']\}$ $(i \leq j \in \mathbb{Q}^{\geq 0})$ such as $\exists([t_i', t_j'], s_1^2) \in T\text{-}Availability(cur^2, m)$ ;
                $Candidates := Candidates \cup \{(s_1^1, s_1^2)\}$ ;
2               $T\text{-}Availability((cur^1, cur^2), m) := T\text{-}Availability((cur^1, cur^2), m) \cup \{([t_k, t_l], (s_1^1, s_1^2))\}$ for
                each $[t_k, t_l] \in \{[t_i, t_j]\} \cap \overline{\{[t_i', t_j']\}}$ ;
3           else
                $Candidates := Candidates \cup \{(s_1^1, \mu)\}$ ;
                $T\text{-}Availability((cur^1, cur^2), m) := T\text{-}Availability((cur^1, cur^2), m) \cup \{([t_k, t_l], (s_1^1, \mu))\}$ for
                each $[t_k, t_l] \in \{[t_i, t_j]\}$ ;
        $Candidates := Candidates \setminus \mathcal{S}$ ;
        foreach $(s^1, s^2) \in \mathcal{S} \mid \tau\text{-}closure(s^1) \cap \mathcal{F}^1 \neq \emptyset \wedge \tau\text{-}closure(s^2) \cap \mathcal{F}^2 = \emptyset$ do
            $\mathcal{F} := \mathcal{F} \cup \{(s^1, s^2)\}$
    return $\mathcal{S}$, $s_0$, $\mathcal{F}$ and for each $s \in \mathcal{S}$, a set $T\text{-}Availability(s, m_e)$ $\forall m_e \in output_e(s)$ ;
end
```

target state of this transition in the resulting protocol is made of the target state of $\mathcal{P}_1$ and a new state, called μ. The rest of the algorithm works in a similar manner as the other algorithms with the difference that final states of the resulting protocol are made of pairs of final states of $\mathcal{P}_1$ and either non final states of $\mathcal{P}_2$ or μ states.
Complexity analysis. Let n and m be respectively the maximal number of states and the maximal number of explicit transitions in the input protocols. Algorithm 3 runs in time $O(n^2m^2)$.

5.4 Timed subsumption algorithm

Algorithm 4 implements a timed subsumption test. The idea of this algorithm is to check if the initial state of $\mathcal{P}_1$ is *simulated*[2] by the initial state of $\mathcal{P}_2$. A state s of $\mathcal{P}_1$ is simulated by a state s' of $\mathcal{P}_2$ if for every outgoing message m in s there is: (i) the same outgoing message m in s' with similar polarity and a similar or wider temporal availability, and (ii) the target state of m in $\mathcal{P}_1$ is simulated by the target state of m in $\mathcal{P}_2$. Moreover, if s is a final state then so must be s'.
Complexity analysis. Let n and m be respectively the maximal number of states and the maximal number of explicit transitions in the input protocols. Algorithm 4 runs in time $O(n^3m)$.

6 Discussion and conclusion

We now briefly compare this work with prior state of the art and summarize our contributions. Several ongoing efforts recognize the need to support the explicit description of business protocols in web services. In the standardization arena, the Business Process Execution Language for Web Services (BPEL), the Web Services Conversation Language (WSCL) and the Web Service Choreography Interface (WSCI), are examples of of specifications that feature support for describing service conversations [16]. Our work provides complementary contributions to these efforts. We focus on abstracting, analyzing, and managing Web services protocols.

As mentioned before, several ongoing efforts in the area of Web services recognize the importance of high level modeling and analysis of service protocols (e.g., [9, 10, 11, 13, 17, 19]). In terms of protocol description, the existing models do not explicitly take important service abstractions such as timed transitions into account. In terms of protocols analysis, mechanisms are proposed to compare verify protocols compatibility and replaceability. Similar approaches for protocols compatibility and replaceability exist in the area of component-based systems [12, 20]. In the above approaches, the proposed techniques do not cater for timed business protocols.

There are some similarities between our work and timed automata [2], a specification formalism which was introduced to enable explicit modeling of time for reactive systems. Several formal aspects such as reachability, language inclusion and equivalence has been investigated for timed automata [3]. It should be noted that timed automata can be extended to provide formal semantics for our model as, for example, in the original timed automata model messages polarity is not supported.

In this paper, we build upon our earlier work on service protocols modeling, analysis, and management [5, 9] to cater for temporal abstractions in business protocols. We can summarize our contributions in terms of protocols modeling, analysis, and management:

- We motivate the need for modeling of timed protocols and for the introduction of formal models and algebras for protocol modeling and management

- We proposed a state-machine model for representing timed protocols. This model is simple and has a formal se-

[2] In fact, our algorithm uses a kind of graph simulation test in which the simulation relation [14] is extended to cater for temporal constraints.

Algorithm 4: A timed subsumption algorithm.

```
Data: Two protocols $\mathcal{P}_1 = (\mathcal{S}^1, s_0^1, \mathcal{F}^1, \mathtt{M}^1, \mathcal{R}^1)$ and $\mathcal{P}_2 = (\mathcal{S}^2, s_0^2, \mathcal{F}^2, \mathtt{M}^2, \mathcal{R}^2)$.
Result: true if $\mathcal{P}_1 \lesssim_T \mathcal{P}_2$, false otherwise.
begin
    $layer_1 := \{s_0^1\}$, $layer_2 := \{s_0^2\}$, $done := \emptyset$ ;
    while $layer_1 \neq \emptyset$ do
        foreach $s^1 \in layer_1$ do
            $found := false$ ;
1           foreach $s^2 \in layer_2$ do
                if $output_e(s^1) \nsubseteq output_e(s^2)$ then continue (1) ;
                foreach $m_e \in output_e(s^1)$ do
                    Let $(i^1, t^1) := T\text{-}Availability(s^1, m_e)$ and $(i^2, t^2) := T\text{-}Availability(s^2, m_e)$ ;
2                   if $(i^1, t^1) \notin_T (i^2, t^2)$ or $(t^1 \in \mathcal{F}^1 \wedge t^2 \notin \mathcal{F}^2)$ or $(t^1 \notin \mathcal{F}^1 \wedge t^2 \in \mathcal{F}^2)$ then continue
                    (1) ;
                $found :=$ true, $done := done \cup \{s^1\}$ ;
3           if $found =$ false then return false ;
            $done := done \cup \{s^1\}$ ;
        $layer_1 := succ(layer_1) \setminus done$, $layer_2 := succ(layer_2)$ ;
    return true ;
end
```

mantics. Both simplicity and formal semantics are necessary to provide effective protocol analysis techniques.

- We provide a formal characterization of the notions of compatibility and replaceability for timed business protocols, emphasizing the subtle issues that may arise in performing this kind of analysis and discussing the need for new replaceability and compatibility classes.

- We proposed a number of operators for analyzing and managing timed business protocols.

Finally, we mention that the work presented in this paper part of a larger framework supported by a CASE tool, partially implemented, that manages the entire service development lifecycle. Our objective of this framework is to provide a comprehensive methodology and platform that can facilitate large-scale interoperation of Web services and substantially reduce the service development effort.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures, and Applications.* Springer Verlag, 2004.

[2] R. Alur. Timed Automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22, Trento, Italy, July 6-10, 1999. Springer.

[3] R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *4th Intl. School on Formal Mthods for Computer, Communication, and Software Systems: Real Time*, pages 449–467, 2004.

[4] 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.

[5] B. Benatallah, F. Casati, and F. Toumani. Web Service Conversation Modeling: A Cornerstone for e-Business Automation. *IEEE Internet Computing*, 6(1), 2004.

[6] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual Modeling of Web Service Conversations. In *Procs of CAiSE'03*, volume 2681 of *LNCS*, pages 449–467, Klagenfurt, Austria, 2003. Springer.

[7] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. On temporal abstractions of web services protocols. In *CAiSE Forum 2005. Porto, Portugal*, June 2005.

[8] Boualem Benatallah, Fabio Casati, Julien Ponge, and Farouk Toumani. Timed web services protocols: compatibility and replaceability analysis (extended version). Technical report, LIMOS Clermont-Ferrand, 2005. `http://www.isima.fr/ponge/research.shtml`.

[9] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and management of web services protocols. In *Procs of ER'04, Shanghai, China*, 2004.

[10] L. Bordeaux, G. Salaun, D. Berardi, and M. Marcella. When are two Web Services Compatible? In *VLDB TES'04, Toronto, Canada*, 2004.

[11] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW 2003, Budapest, Hungary*, pages 403–410. ACM, May 2003.

[12] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo. Adding Roles to CORBA Objects. *IEEE Trans. Software Eng*, 29(3):242–260, March 2003.

[13] X. Dong, A.Y. Halevy, J. Madhavan, E. Nemes, and J Zhang. Similarity Search for Web Services. In *VLDB'04. Toronto, Canada*, 2004.

[14] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum (extended abstract). In *CONCUR'90*, volume 458 of *LNCS*, pages 278–297. Springer, 1990.

[15] Richard Hull, Michael Benedikt, Vassilis Christophides, and Jianwen Su. E-services: a look behind the curtain. In *Proc. 22th Principles of Database Systems (PODS'03), San Diego, CA, USA*, pages 1–14. ACM, June 2003.

[16] M. P. Papazoglou and D. Georgakopoulos. Special issue on service oriented computing. *Commun. ACM*, 46(10):24–28, 2003.

[17] S. R. Ponnekanti and A. Fox. Interoperability among Independently Evolving Web Services. In *Middleware '04. Toronto, Canada, 2004*, 2004.

[18] S. Vinoski. W WS-Nonexistent Standards. *IEEE Internet Computing*, 8(6):94–96, 2004.

[19] A. Wombacher, B. Mahleko, P. Fankhauser, and E. Neuhold. Matchmaking for Business Processes based on Choreographies. In *EEE'04. March 2004, Taipei, Taiwan*, 2004.

[20] D.M. Yellin and R.E. Storm. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, March 1997.