Department of Languages and Computer Science
University of Málaga

Measuring the Compatibility of Service Interaction Protocols

[1] **Meriem Ouederni,** [2] **Gwen Salaün, and** [1] **Ernesto Pimentel**

[1] University of Málaga, Spain
[2] Gr. INP–INRIA–LIG, France
Email [1] {meriem,ernesto}@lcc.uma.es
[2] Gwen.Salaun@inria.fr

Technical Report ITI 4-10

October 2010

Abstract

Checking the compatibility of service interfaces allows one to avoid erroneous executions when composing services together. This task is especially difficult when considering interaction protocols in service interfaces. Although the compatibility verification has been intensively studied, in particular for discovery purposes, most of existing work return a Boolean result. However, if two services are incompatible, these approaches do not indicate whether the services are almost compatible or totally incompatible. This information is crucial if one wants to apply adaptation techniques, for instance, to successfully compose these services in spite of existing mismatches. In this paper, we propose a flooding-based approach for measuring the compatibility degree of service interfaces. We illustrate our approach with two compatibility notions, namely *unspecified receptions* and *unidirectional complementarity*. Our proposal is fully automated by a prototype tool we have implemented.

Keywords: Service Interfaces, Interaction Protocols, Formal Verification, Compatibility Flooding.

1 Introduction

Service Oriented Computing (SOC) promotes the construction of new systems by reusing existing software units called services. These services are developed independently and accessed through their interfaces which distinguish several interoperability levels (*i.e.*, signature, interaction protocol, quality of service, and semantics). A key issue is to check whether the service interfaces are compatible or not. The goal of this check is to guarantee the successful interoperation of services. In this report, we focus on the interaction protocol level of service interfaces. Checking the compatibility of interaction protocols is a tedious and hard task even though this is of utmost importance to avoid run-time errors, *e.g.,* deadlock situations or unmatched messages. Most of the existing approaches (see for instance [10, 33, 2, 17, 12, 4]) return a "True" or "False" result to detect whether services are compatible or not. Unfortunately, a Boolean answer is not very helpful for many reasons. First, in real world case studies, there will seldom be a perfect match, and when service protocols are not compatible, it is useful to differentiate between services that are slightly incompatible and those that are totally incompatible. Furthermore, a Boolean result does not give a detailed measure of which parts of service protocols are compatible or not.

To overcome the aforementioned limits, a new solution aims at measuring the compatibility degree of service interfaces. This issue has been addressed by a few recent works, see for instance [32]. However, most of them are based upon description models of service interfaces which do not consider value-passing coming with exchanged messages and internal behaviours (τ transitions). Internal behaviours in interface models are very important because some services can be compatible from an observable point of view, but their execution will behave erroneously if these behaviours are not taken into account (see Section 2 for a detailed discussion). Moreover, existing approaches measure the interface compatibility using a simple (*i.e.*, not iterative) traversal of protocols, and consider a unique compatibility notion making their application quite restricted.

In this report, we propose a generic framework where the compatibility degree of service interfaces can be automatically measured according to different compatibility notions. We illustrate our approach using a bidirectional and an unidirectional compatibility notions, namely *unspecified receptions* and *unidirectional complementarity*. Additional compatibility notions can easily be added to our framework. We consider a formal model for describing service interfaces with interaction protocols (messages and their application order, but also value-passing and internal actions). In our approach, the compatibility is computed in two steps. A first step computes a set of static compatibility degrees where the execution order of messages is not taken into account. Then, a flooding algorithm computes the compatibility degree of interaction protocols using the static compatibility results. The computation process also returns the mismatch list indicating the interoperability issues. The proposed framework is fully automated by a prototype tool (called Comparator) we have implemented.

Our compatibility measure brings more advantages than the Boolean approaches and this opens a wide range of applications, in particular automatic service adaptation [22]. If a set of services are incompatible, the detailed measures and the mismatch list help to understand what parts of these services do not match. Thus, the mismatches can be worked out using adaptation techniques, and service composition can be achieved in spite of existing mismatches.

The remainder of this report is structured as follows. Section 2 describes our model of services. Section 3 introduces the compatibility notions we use in this report for illustration purposes. In Section 4, we present our solution for measuring the service compatibility. Section 5 introduces our prototype tool and some experimental results. Section 6 states a brief comparison with related approaches. Finally, Section 7 draws some conclusions.

2 Service Model

We assume service interfaces are described using their interaction protocols represented by *Symbolic Transition Systems* (STSs). Our STS is a variant of STG (Symbolic Transition Graph) presented in [18], where guards are abstracted here as transitions labelled with τ actions. Communication between services is represented using *events* relative to the emission and reception of messages. Events come with a list of typed parameters (possibly empty). In our model, a *label* is either the (internal) τ action or a tuple (m, d, pl) where m is the message name, d stands for the communication direction (either an emission ! or a reception ?), and pl is either a list of typed data terms if the label corresponds to an emission (output action), or a list of typed variables if the label is a reception (input action).[1]

Definition 1 (STS) *A Symbolic Transition System, or STS, is a tuple* (A, S, I, F, T) *where:* A *is an alphabet which corresponds to the set of labels associated to transitions,* S *is a set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *is a nonempty set of final states, and* $T \subseteq S \backslash F \times A \times S$ *is the transition relation.*

This model is simple yet offers a good abstraction level for describing and analysing service behaviours. Moreover, STSs can be easily derived from abstract descriptions implemented in existing platform languages (*e.g.*, Abstract BPEL or WF), see for instance [14, 29, 13, 5] where such abstractions for Web services were used for verification, composition or adaptation purposes. For the sake of clarity, in the rest of the article, we will describe service interfaces only with their corresponding STSs. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels.

2.1 STS Operational Semantics

The operational semantics of one STS ($\rightarrow_b$) is defined with three rules for, respectively, internal action (TAU), emission (EM), and reception (REC) given in Figure 1. The pair $\langle s_i, E_i \rangle$ consists of an active state[2] $s_i \in S_i$ and a data environment E_i. A data environment is a set of pairs $\langle x, v \rangle$ where x is a variable and v a ground value. The environment can be updated using the operator "$\oslash$" which assigns a new ground value to an existing variable. The update operator can also add a new variable and its ground value to the environment:

$$(E \cup \{\langle x, v \rangle\}) \oslash \langle x', v' \rangle \triangleq E \cup \{\langle x, v' \rangle\} \; if \; x = x'$$
$$(E \cup \{\langle x, v \rangle\}) \oslash \langle x', v' \rangle \triangleq (E \oslash \langle x', v' \rangle) \cup \{\langle x, v \rangle\} \; if \; x \neq x'$$
$$\emptyset \oslash \langle x, v \rangle \triangleq \{\langle x, v \rangle\}$$

The data evaluation operator "ev" is defined as follows:

$$ev(E, x) \triangleq E(x)$$
$$ev(E, f(v_1, \ldots, v_n)) \triangleq f(ev(E, v_1), \ldots, ev(E, v_n))$$

where the function $E(x)$ returns the value of x in the environment:

$$(E \cup \langle x, v \rangle)(x') \triangleq v \; if \; x = x'$$
$$(E \cup \langle x, v \rangle)(x') \triangleq E(x') \; if \; x \neq x'$$

Notice that, using the STS model, a choice can be represented using either a state and at least two outgoing transitions labelled with observable actions (external choice) or branches of τ actions (internal choice).

The operational semantics of n STSs $STS_{i \in \{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$ ($\rightarrow_c$) is formalised using the synchronous communication[3] rule COM and the independent evolution rule INE$_\tau$ given in Figure 2. The function *get-type* returns the type of a variable or value, and $\{as_1, \ldots, as_n\}$ denotes a set of active states. In the COM rule, value passing and variable substitutions rely on a late binding semantics [24].

[1] The message names and parameter types respect the service signature.

[2] We assume that the state identifiers are disjoint.

[3] Although checking protocol compatibility is undecidable with asynchronous communication [3], Fu *et al.* proved in [15] that a large class of interfaces can be analysed under an asynchronous communication model using techniques and tools existing for the synchronous communication model.

$$\frac{s \xrightarrow{\tau} s'}{\langle s, E\rangle \xrightarrow{\tau}_b \langle s', E\rangle} \quad \text{(TAU)}$$

$$\frac{s \xrightarrow{a!e} s' \quad v' = ev(e, E)}{\langle s, E\rangle \xrightarrow{a!v'}_b \langle s', E\rangle} \quad \text{(EM)}$$

$$\frac{s \xrightarrow{a?x} s'}{\langle s, E\rangle \xrightarrow{a?x}_b \langle s', E\rangle} \quad \text{(REC)}$$

Figure 1: Operational semantics of a STS.

$$\frac{\begin{array}{c} i, j \in \{1..n\} \quad i \neq j \\ \langle s_i, E_i\rangle \xrightarrow{a!v}_b \langle s_i', E_i\rangle \quad \langle s_j, E_j\rangle \xrightarrow{a?x}_b \langle s_j', E_j\rangle \\ \textit{get-type}(x) = \textit{get-type}(v) \quad E_j' = E_j \oslash \langle x, v\rangle \end{array}}{\{as_1, .., \langle s_i, E_i\rangle, .., \langle s_j, E_j\rangle, .., as_n\} \xrightarrow{a!v}_c \{as_1, .., \langle s_i', E_i\rangle, .., \langle s_j', E_j'\rangle, .., as_n\}} \quad \text{(COM)}$$

$$\frac{i \in \{1..n\} \quad \langle s_i, E_i\rangle \xrightarrow{\tau}_b \langle s_i', E_i\rangle}{\{as_1, .., \langle s_i, E_i\rangle, .., as_n\} \xrightarrow{\tau}_c \{as_1, .., \langle s_i', E_i\rangle, .., as_n\}} \quad (\text{INE}_\tau)$$

Figure 2: Operational semantics of n STSs.

Internal Behaviours. Service analysis could be worked out without taking into account their internal evolution because that information is not observable from its partners point of view (black-box assumption). However, keeping an abstract description of the non-observable behaviours while analysing services helps to find out possible interoperability issues. Indeed, although one service can behave as expected by its partner from an observable point of view, interoperability issues may occur because of unexpected internal behaviours that services can execute. For instance, Figure 3 shows two service protocols (Client and Client') exhibiting the same observable behaviour. However, Client' gives information about internal decisions whereas Client does not. The Client and the Server can interoperate on update and terminate in final states (register! in Client has no counterpart in Server and cannot be executed due to the synchronous communication semantics). However, if we consider Client', which is an abstraction closer to what the service actually does, we see that this protocol can (choose to) execute a τ transition in state s1 and arrives in state s3 while Server is still in state u1. At this point, both Client' and Server cannot exchange messages, and the system deadlocks. This issue would not have been detected with Client.

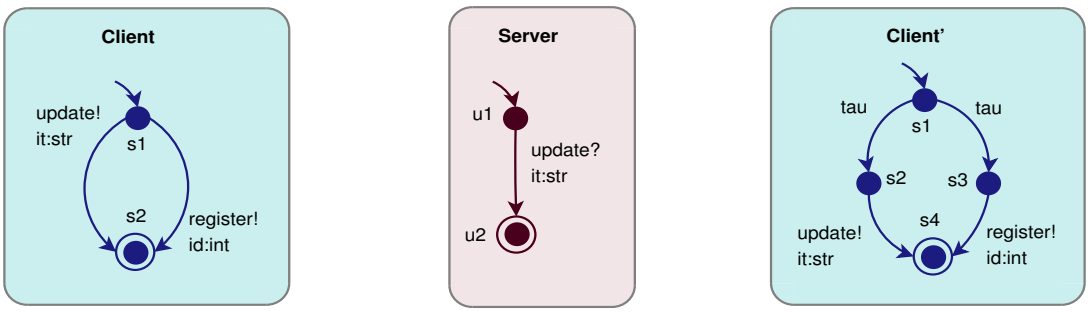


Figure 3: Internal Behaviours in Service Protocols.

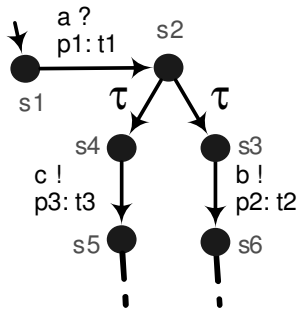
Now, let us focus on higher-level languages, such as abstract BPEL or abstract Windows workflow (WF), which are used in the literature [22, 9, 21] as abstract descriptions (Interface Description Languages) of service behaviours. Here we choose WF to illustrate how STSs and in particular τ transitions are extracted from this workflow-based notation. WF describes service behaviours using a set of basic activities, *e.g.*, IfElse, Listen and While, for which it is useful to keep some τ transitions in their respective STS descriptions.

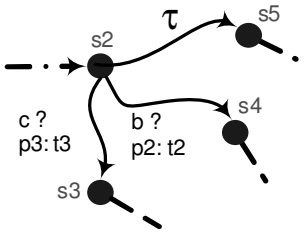
The IfElse activity corresponds to an internal choice deciding which activity has to be performed, *e.g.*, sending different messages using the WebServiceOutput activity, depending on the condition truth value. The corresponding STS contains as many transitions labelled with τ as there are branches in the IfElse activity (including the *else* branch), see the first example in Table 1.

Transitions labelled with τ can describe timeouts, as it is the case in the Listen activity of WF. This activity waits for possible receptions (EventDriven). If no message is received, a timeout occurs (Delay) which stops the Listen activity. In the STS model, the Listen activity is translated into a set of branches labelled with the receptions used in this activity and a τ transition corresponding to the timeout, see the second example in Table 1.

The While activity is used to repeat an activity as long as the loop condition is satisfied. Hence, the corresponding STS encodes this activity using a non-deterministic choice, specified using τ transitions, between the looping behaviour and the behaviour that can be executed after the While activity (when the condition becomes false), see the third example in Table 1.

Abstract WF activity	STS description

`WebServiceInput(a?(p1:t1))`
`ifElse`
`(((p1 < 10),`
`WebServiceOutput(b!(p2:t2))),`
`((p1 ≥ 10),`
`WebServiceOutput(c!(p3:t3)))`
`)`
`...`
	

`...`
`listen(`
`EventDriven`
`(WebServiceInput(b?(p2:t2)),...),`
`EventDriven`
`(WebServiceInput(c?(p3:t3)),...),`
`EventDriven(Delay,...)`
`)`
	

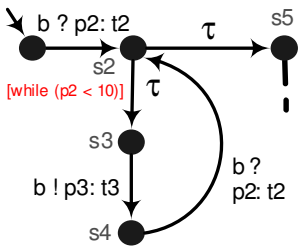
`WebServiceInput(b?(p2:t2))`
`While`
`(`
`(p2 < 10),`
`InvokeWebService`
`(b!(p3:t3), b?(p2:t2))`
`)`
`...`
	

Table 1: Examples of Abstract WF Activities and their Corresponding STSs

Other abstract WF activities such as Terminate, Parallel and Code can also generate τ transitions in the corresponding STS model.

3 Protocol Compatibility

Compatibility checking verifies the successful interaction between services *wrt.* a criterion set on their observable actions. This criterion is referred to as a compatibility notion. In this report, we distinguish two classes of notions depending on the direction of the compatibility checking. We refer to these classes as bidirectional and unidirectional checking. We particularly illustrate our approach with a bidirectional compatibility notion, namely *unspecified receptions* (*UR* for short), and with an unidirectional notion, namely *unidirectional complementarity* (*UC* for short).

3.1 Preliminaries

This section introduces some basic concepts needed to define the *UR* and *UC* compatibility notions. In what follows, we describe a transition using a tuple (s, l, s') such that s and s' denote the source and target states, respectively, and l stands for its label. We suppose that for all transitions (s, τ, s'), $s \neq s'$. Given two services described using STSs, $STS_{i\in\{1,2\}} = (A_i, S_i, I_i, F_i, T_i)$, we define a global state as a pair of states $(s_1, s_2) \in S_1 \times S_2$. For the sake of comprehension, we have chosen to present several simple examples instead of a single running example. However, we have applied our approach to many real-world case studies. Some of them are mentioned in Section 5 and other ones are available on-line at [26]. For clarity purposes, we assume in the rest of this report that the different functions defined have access to the $STS_{i\in\{1,2\}} = (A_i, S_i, I_i, F_i, T_i)$ even if they are not explicitly passed as input parameters. However, we made parameters explicit if they are modified.

Parameter Compatibility. The usual meaning of parameter compatibility requires that the parameter list expected to be received perfectly matches (same types in the same order) the parameter list coming with the sent message. We refer to this definition as *parameter matching* (*pm* for short).

Definition 2 (Parameter Matching) *Two parameter lists $pl_1 = (p_{11}, \ldots, p_{1n})$ and $pl_2 = (p_{21}, \ldots, p_{2m})$ are compatible, $par\text{-}match(pl_1, pl_2)$, iff $n = m$, and $\forall k \in \{1, \ldots, n\},\ get\text{-}type(p_{1k}) = get\text{-}type(p_{2k})$.*

Notice that more sophisticated strategies may similarly be defined, *e.g.*, it is possible to receive less parameters than those being sent, the parameters can be sent and received in a different order, or the types do not need to coincide (subtyping, automatic conversion, etc.).

Label Compatibility. Label comparison is necessary to check whether exchanged messages and their parameters are compatible. Two labels are considered compatible if they have opposite directions, same names, and compatible parameters. This definition is called *label matching* and formalised as follows:

Definition 3 (Label Matching) *Two labels l_1 and l_2 are compatible, $lab\text{-}match(l_1, l_2)$, iff:*

- *$l_1 = (m_1, d_1, pl_1)$ and $l_2 = (m_2, d_2, pl_2)$, $m_1 = m_2$, $d_1 = \overline{d_2}$ and $par\text{-}match(pl_1, pl_2)$, or*
- *$l_1 = \tau$ and $l_2 = \tau$,*

where $\overline{!} = ?$, $\overline{?} = !$.

Reachable States. Reachability analysis aims at computing the set of global states that interacting protocols can access, in zero or more steps, from a current global state (s_1, s_2). Protocols can move into reachable states through synchronisations on compatible labels or independent evolutions, *i.e.*, τ transitions.

Definition 4 (Reachable States) *The function $reachable((s_1, s_2))$ returns the smallest set of global states reachable from (s_1, s_2) such that:*
$\forall (s_1, l_1, s_1') \in T_1$:

- *if $l_1 = \tau$, then:*
  - *$reachable((s_1', s_2)) \subseteq reachable((s_1, s_2))$, and*
  - *if $s_1' \in F_1$, or $\exists (s_1', l_1', s_1'') \in T_1,\ l_1' \neq \tau$, then $(s_1', s_2) \in reachable((s_1, s_2))$.*
- *else, $\forall (s_2, l_2, s_2') \in T_2$ with $lab\text{-}match(l_1, l_2)$, $\{(s_1', s_2')\} \cup reachable((s_1', s_2')) \subseteq reachable((s_1, s_2))$.*

Additionally, if $(s_2', s_1') \in reachable((s_2, s_1))$, then $(s_1', s_2') \in reachable((s_1, s_2))$

Example 1 *Figure 4 shows an example of two service protocols, which allows to make an update in a database once a user account is created. As we can observe, the protocols can initially transit from* $(\mathsf{s1}, \mathsf{c1})$ *to state* $(\mathsf{s2}, \mathsf{c2})$ *through the compatible labels* register?id:int *and* register!id:int. *However, both protocols cannot synchronise on the update message because* update? *is not compatible with any label in* c1. *Applying the same reasoning on* $(\mathsf{s2}, \mathsf{c2})$*, the set of global states reachable from the initial one is* $\{$*(*s2,c2*),(*s1,c3*),(*s3,c4*)*$\}$.

Deadlock-Freeness. An important property required for checking the successful system termination is deadlock-freeness. In order to check that services can always interoperate starting from a given global state until reaching final states, we define deadlock-freeness as follows:

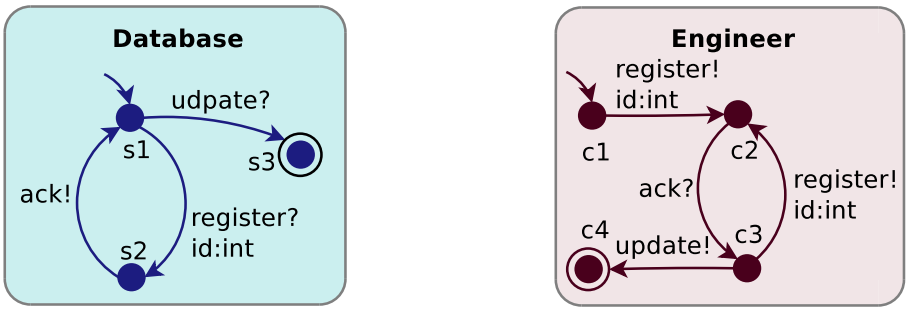


Figure 4: Database Handling System (I).

Definition 5 (Deadlock-Freeness) *Given two STSs* $STS_{i\in\{1,2\}} = (A_i, S_i, I_i, F_i, T_i)$*, the set of their deadlock-free global states,* df*, is the least set such that* $(s_1, s_2) \in df$ *if and only if either* $(s_1, s_2) \in (F_1 \times F_2)$ *or* $\forall (s'_1, s'_2) \in reachable((s_1, s_2))$*,* $s'_1 \neq s_1$ *and* $s'_2 \neq s_2$*,* $(s'_1, s'_2) \in df$*.*

State Compatibility. Service interaction basically depends on synchronisations over observable actions and then can be defined using a criterion set on them. The criterion is used to check the state compatibility as follows. For a given global state (s_1, s_2), this state is considered compatible if every the message l_1 sent (received, respectively) by protocol 1 at state s_1 will be eventually received (sent, respectively) by protocol 2 at state s_2, such that both protocols evolve into a compatible global state, and vice-versa. If protocol 2 is not able to interact with protocol 1's action, then both protocols must be able to reach a global state (s_1, s'_2) in which this action will be satisfied, *i.e.*, $\exists (s'_2, l_2, s''_2) \in T_2$ such that l_1 and l_2 are compatible, and vice-versa. In this case, the protocols must also be compatible in (s_1, s'_2) and (s'_1, s''_2). Since services can evolve independently through some τ transitions, the behavioural compatibility requires that *each* internal evolution must lead both services into compatible states [7, 10]. This means that every time a τ transition is traversed in one protocol, then the compatibility has to be checked again on the target state. We refer to this compatibility definition as *state matching* which is formally defined as follows:

Definition 6 (State Matching) *Given two STSs,* $STS_{i\in\{1,2\}} = (A_i, S_i, I_i, F_i, T_i)$*, and a label direction* d*, the set of* d*-compatible states,* $state\text{-}match_d$*, is the largest set such that if a global state* $(s_1, s_2) \in state\text{-}match_d$ *then:*

- $(s_1, s_2) \in state\text{-}match_{d,\rightarrow}(s_1, s_2)$
- $(s_2, s_1) \in state\text{-}match_{d,\rightarrow}(s_2, s_1)$

where $\exists i, j \in \{1, 2\},\ i \neq j$*, and* $state\text{-}match_{d,\rightarrow}(s_i, s_j)$ *is the largest set such that* $\forall (s_i, l_i, s'_i) \in T_i$*, if* $(s_i, s_j) \in state\text{-}match_{d,\rightarrow}(s_i, s_j)$*, then:*

- *if* $l_i = (m_i, d, pl_i)$*, then either*
  - $\exists (s_j, l_j, s'_j) \in T_j$ *such that* $lab\text{-}match(l_i, l_j)$ *and* $(s'_i, s'_j) \in state\text{-}match_d$*, or*
  - $\exists (s_i, s'_j) \in reachable((s_i, s_j))$ *such that* $\exists (s'_j, l'_j, s''_j) \in T_j$ *where:*
    - $lab\text{-}match(l_i, l'_j)$*,*
    - $(s_i, s'_j) \in state\text{-}match_d$*, and*
    - $(s'_i, s''_j) \in state\text{-}match_d$*.*
- *else if* $l_i = \tau$*, then* $(s'_i, s_j) \in state\text{-}match_d$*.*

Example 2 *Let us show that the global state* $(\mathsf{s1}, \mathsf{c1})$ *is in the set* $state\text{-}match_? \cup state\text{-}match_!$ *obtained for protocols* Database *and* Engineer *in Figure 4. Although the label* register?id:int *at state* s1 *can match the label* register!id:int *at state* c1*, this is not the case of the label* update? *at state* s1 *because it does not match any label at state* c1*. However, both STSs are able to reach the global state* $(\mathsf{s1}, \mathsf{c3})$ *in which the label* update? *can be matched. Furthermore,* Database *and* Engineer *are compatible in* $(\mathsf{s1}, \mathsf{c3})$ *and also in* $(\mathsf{s3}, \mathsf{c4})$*. As we can observe, every synchronisation leads both STSs into compatible global state, therefore* $(\mathsf{s1}, \mathsf{c1}) \in (state\text{-}match_? \cup state\text{-}match_!)$*.*

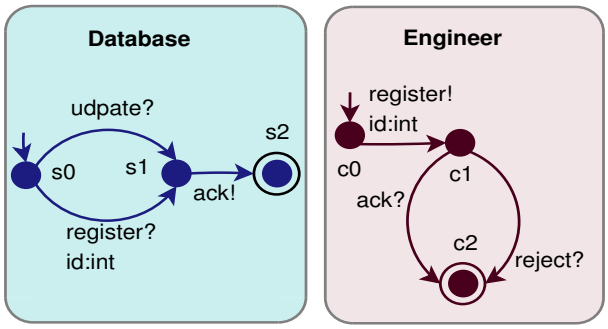


Figure 5: Database Handling System (II).

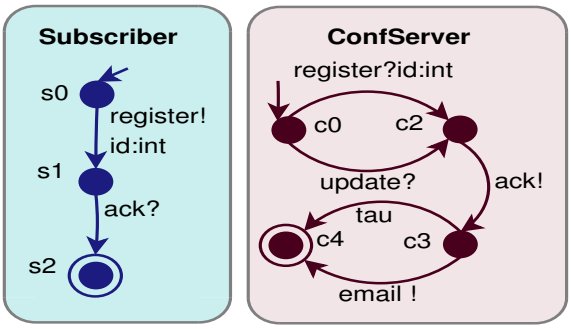


Figure 6: Conference Registration System.

3.2 Notions of Protocol Compatibility

Unspecified Receptions (*UR*). This notion is inspired from [33] and requires that two services are compatible (i) if they are deadlock-free at their initial global state, and (ii) if one service can send a message at a reachable state, then its partner must eventually receive that emission such that both services evolve into a compatible global state. The second condition is checked using the verification of state compatibility over the emission transitions. In real-life cases, one service must receive all requests from its partner, but can also be ready to accept other receptions, since the service could interoperate with other partners. Hence, there might be additional unmatched receptions in reachable states, possibly, followed by unmatched emissions. These emissions do not give rise to an incompatibility issue as long as their source states are unreachable when protocols interact with each other.

Definition 7 (Unspecified Receptions) *Two STSs,* $STS_{i\in\{1,2\}} = (A_i, S_i, I_i, F_i, T_i)$*, are UR compatible iff:*

- $(I_1, I_2) \in$ *state-match*$_!$*, and*
- $(I_1, I_2) \in df$.

Example 3 *Let us illustrate the verification of the UR compatibility on the* Engineer *and* Database *protocols in Figure 5. At the initial global state* (s0, c0)*, there is a unique emission,* register!id:int*, which perfectly matches with* register?id:int*. There is also an unmatched reception* update? *at state* s0 *but this does not raise an incompatibility issue according to the above definition. At the global state* (s1, c1)*, the unique emission* ack! *perfectly matches with* ack?*, and here again there is an additional reception* reject?*. Moreover, these protocols do not deadlock. As a result, they are compatible wrt. the UR notion.*

Unidirectional Complementarity (*UC*). Two services are compatible *wrt.* the *UC* notion if there is one service (*complementer*) which must eventually receive (send, respectively) all messages that its partner (*complemented*) expects to send (receive, respectively) at all global reachable states. In addition, both services must be deadlock-free in all reachable global states. Hence, the *complementer* service may send and receive more messages than the *complemented* service.[4] This asymmetric notion is useful to check the successful communication in the client/server model where a server can interact with clients having different behaviours. In this setting, each client behaviour must be satisfied (complemented) by the server.

Definition 8 (Unidirectional Complementarity) *An* STS_{er} *complements another one* STS_{ed} *iff* $\exists T'_{er} \subseteq T_{er}$ *such that for* $STS_{er} = (A_{er}, S_{er}, I_{er}, F_{er}, T'_{er})$ *and* $STS_{ed} = (A_{ed}, S_{ed}, I_{ed}, F_{ed}, T_{ed})$*:*

- $(I_{er}, I_{ed}) \in$ *state-match*$_!$*,*
- $(I_{er}, I_{ed}) \in$ *state-match*$_?$*, and*
- $(I_{er}, I_{ed}) \in df$.

Example 4 *Figure 6 consists of two protocols: the* Subscriber *(complemented) first asks for a conference registration and waits for an acknowledgment. The conference server* ConfServer *(complementer) can receive a*

[4]Our definition is different than the usual simulation or preorder [8] relation since we compare protocols with opposite directions.

request for either a registration or an updating. Then, the server sends back to the subscriber an acknowledgement followed by a confirmation email, or terminates if this confirmation has not to be sent (described with a τ transition). We notice that the ConfServer *complements the* Subscriber *because every time the* Subscriber *wants to transit into another state the* ConfServer *enables that transition. Moreover, both protocols are free of deadlocks. Although there is an unmatched emission* email! *in the reachable global state* $(\mathsf{s2}, \mathsf{c3})$*, the protocols remain compatible wrt. the UC notion, because this emission is in the complementer protocol. However, they are not compatible wrt. the UR notion because of this reachable but unmatched emission.*

4 Measuring Protocol Compatibility

This section presents our techniques for measuring the compatibility of two service protocols. All the compatibility measures we present in the sequel belong to [0..1] where 1 means a perfect compatibility. The approach overviewed in Figure 7 consists first in computing a set of static compatibility measures (Section 4.1). In a second step, these static measures are used for computing the behavioural compatibility degree for all global state in $S_1 \times S_2$ (Section 4.2). Last, the result is analysed and a global compatibility degree is returned (Section 4.3).

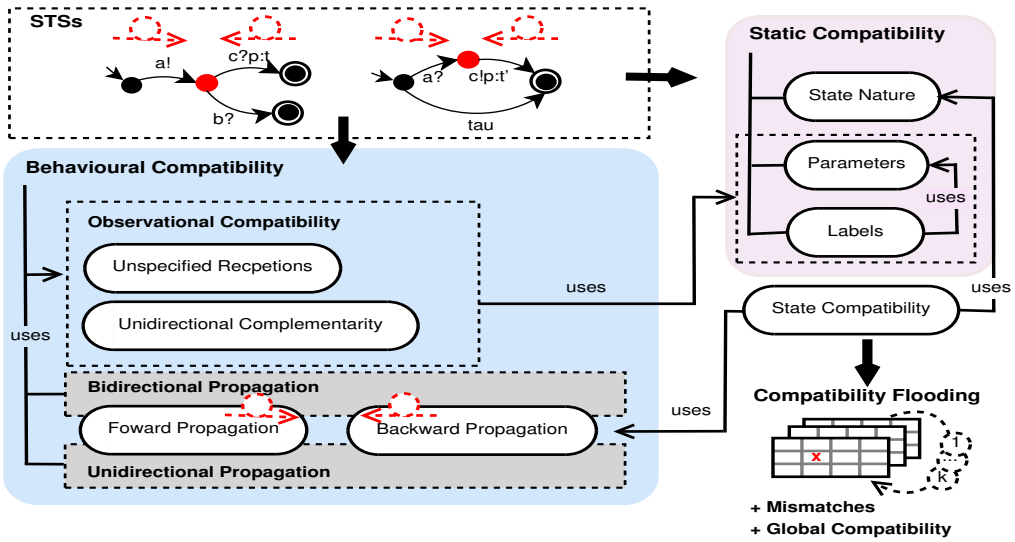


Figure 7: Compatibility Measuring Process.

4.1 Static Compatibility

We use three static compatibility measures, namely state natures, labels, and exchanged parameters.

State Nature. To compare state nature, we use the function $nat(s_1, s_2)$ which returns 1 if the two given states s_1 and s_2 have the same nature, *i.e.,* both states are either initial, final or none of them. Otherwise, $nat(s_1, s_2)$ returns 0.

$$nat(s_1, s_2) = \begin{cases} 1 & \text{if } (s_1, s_2) = (I_1, I_2) \\ & \vee\ (s_1, s_2) \in F_1 \times F_2 \\ & \vee\ ((s_1, s_2) \neq (I_1, I_2)) \wedge ((s_1, s_2) \notin F_1 \times F_2) \\ 0 & \text{otherwise} \end{cases}$$

Parameters. According to Definition 2, the compatibility degree of two parameter lists pl_1 and pl_2 depends on three auxiliary measures, namely: (i) the compatibility of parameter number computed using the function $number$. This function compares the difference between the length of the parameter lists pl_1 and pl_2; (ii) the compatibility of parameter order measured with the function $order$, and (iii) the compatibility of parameter type determined using the function $type$. These measures must be set to 1 if $pl_1 \cup pl_2 = \varnothing$. Otherwise, they are computed as follows: First, we compute the score of the respective mismatch, *i.e.,* different lengths of parameter lists, unordered types and/or unshared types in both parameter lists. Then, we normalise the score by the maximal

value that can be achieved. Finally, we decrease the mismatch score from the perfect compatibility degree (1) to get the final compatibility degree. For instance, to compute the compatibility degree of parameter number $number(pl_1, pl_2)$, the respective mismatch score consists of the absolute value of the difference between the lengths of parameter lists ($abs(\|pl_1\| - \|pl_2\|)$) normalised by the maximum which is the maximal length between those of pl_1 and pl_2 ($max(\|pl_1\|, \|pl_2\|)$).

$$number(pl_1, pl_2) = 1 - \frac{abs(\|pl_1\| - \|pl_2\|)}{max(\|pl_1\|, \|pl_2\|)}$$

In order to compute $order(pl_1, pl_2)$, we use a function $unorderedTypes$ which returns the set of parameter types existing in both parameter lists, *i.e.*, shared types, but not in the same order from left to right. The set of shared types is computed using a function $sharedTypes(pl_1, pl_2)$.

$$order(pl_1, pl_2) = 1 - \frac{\|unorderedTypes(pl_1, pl_2)\|}{\|sharedTypes(pl_1, pl_2)\|}$$

Last, to compute $type(pl_1, pl_2)$, we need a function $unsharedTypes$ which returns the set of parameter types existing in one parameter list but not in the other.

$$type(pl_1, pl_2) = 1 - \frac{\|unsharedTypes(pl_1, pl_2)\|}{\|pl_1\| + \|pl_2\|}$$

The function $par\text{-}comp$ computes the parameter compatibility as the average of the measures returned by the three previous functions:

$$par\text{-}comp(pl_1, pl_2) = \frac{number(pl_1, pl_2) + order(pl_1, pl_2) + type(pl_1, pl_2)}{3}$$

Example 5 *Let us consider two parameter lists* $pl_1 = (usr{:}str, pwd{:}int)$ *and* $pl_2 = (log{:}str, sig{:}float, pwd{:}int)$. *We show below the computation of the aforementioned measures:*

- *The number compatibility is equal to* $1 - \frac{3-2}{3} = 0.66$. *In the worst case, a non-empty parameter list can be compared with an empty one. Therefore, the denominator must be set as the maximal size among those of* pl_1 *and* pl_2.
- *The order compatibility is equal to* $1 - \frac{1}{2} = 0.5$ *since* pl_1 *and* pl_2 *have one unordered type among two types existing in both lists.*
- *The type compatibility is equal to* $1 - \frac{1}{5} = 0.8$ *because* pl_2 *does not share the type* $float$ *with* pl_1. *The number of unshared types is normalised with the sum of* pl_1 *and* pl_2 *sizes because in the worst case both lists could have types totally different.*
- *As a consequence, the parameter compatibility is equal to* $\frac{0.66+0.5+0.8}{3} = 0.65$.

Labels. Given a pair of labels $(l_1, l_2) \in A_1 \times A_2$, the function $lab\text{-}comp(l_1, l_2)$ returns 0 if both labels l_1 and l_2 have the same direction. Otherwise, the compatibility measure of l_1 and l_2 is computed as the average of (i) the semantic compatibility of message names (m_1, m_2) computed by the function $sem\text{-}comp$ using the Wordnet similarity package [28], and (ii) the parameter compatibility returned by $par\text{-}comp$:

$$lab\text{-}comp((m_1, d_1, pl_1), (m_2, d_2, pl_2)) = \begin{cases} 0 & \text{if } d_1 = d_2 \\ \frac{sem\text{-}comp(m_1, m_2) + par\text{-}comp(pl_1, pl_2)}{2} & \text{otherwise} \end{cases}$$

Message names and parameters can be compared using other techniques such as the N-gram algorithm [20]. It is also possible to compare the semantics of parameter names and/or types using the Wordnet similarity package.

4.2 Behavioural Compatibility

We consider a flooding algorithm which performs an iterative measuring of behavioural compatibility for every global state in $S_1 \times S_2$. This algorithm incrementally propagates the compatibility between neighbouring states using backward and forward processing. The compatibility propagation is based on the intuition that

two states are compatible if their backward and forward neighbouring states are compatible, where the backward and forward neighbours of global state (s'_1, s'_2) in transition relations $T_1 = \{(s_1, l_1, s'_1), (s'_1, l'_1, s''_1)\}$ and $T_2 = \{(s_2, l_2, s'_2), (s'_2, l'_2, s''_2)\}$ are the states (s_1, s_2) and (s''_1, s''_2), respectively. The flooding algorithm returns a matrix denoted $COMP^k_{CN,D}$ where each entry $COMP^k_{CN,D}[s_1, s_2]$ stands for the compatibility measure of global state (s_1, s_2) at the k^{th} iteration. The parameter CN refers to the considered compatibility notion which must be checked being given D that is either an unidirectional ($\rightarrow$) or a bidirectional ($\leftrightarrow$) protocol analysis. $COMP^0_{CN,D}$ represents the initial compatibility matrix where all states are supposed to be perfectly compatible, *i.e.*, $\forall (s_1, s_2) \in S_1 \times S_2,\ COMP^0_{CN,D}[s_1, s_2] = 1$. Then, in order to compute $COMP^k_{CN,D}[s_1, s_2]$, we need two functions, namely $obs\text{-}comp^k_{CN,D}$ and $state\text{-}comp^k_{CN,D}$ that we detail in the following. The first function computes the compatibility of outgoing (incoming, respectively) observable transitions being given a compatibility notion CN. We refer to this measure as observational compatibility. The second function propagates the compatibility from the forward and backward (denoted fw and bw for short, and illustrated in figure 7 with red dashed arrows) neighbouring states to (s_1, s_2) taking into account τ transitions. Thus, the computation of $state\text{-}comp^k_{CN,D}$ combines two auxiliary functions, namely $fw\text{-}propag^k_{CN,D}$ and $bw\text{-}propag^k_{CN,D}$.

In this report, we only present the forward compatibility, the backward compatibility can be handled in a similar way based upon incoming rather than outgoing transitions. In the following, we start by introducing the computation of observational compatibility *wrt.* to *UR* and *UC* notions presented in Section 3.2. In the following, we first present $obs\text{-}comp^k_{CN,D}$ and $fw\text{-}propag^k_{CN,D}$. Then, we define $state\text{-}comp^k_{CN,D}$. Finally, we introduce the computation of $COMP^k_{CN,D}[s_i, s_j]$.

Before defining $obs\text{-}comp^k_{CN,D}$, we need to present a few functions necessary to its computation. Given a state $s \in S$ and a transition set T, we define the set of emissions, receptions, and forward transitions going out from s, respectively, as follows:

$$E(s,T) = \{t \in T \mid t = (s, (m, !, pl), s')\}$$

$$R(s,T) = \{t \in T \mid t = (s, (m, ?, pl), s')\}$$

$$Fw(s,T) = E(s,T) \cup R(s,T)$$

We also refer to $tau(s,T)$ as the set of transitions labelled with τ actions and going out from the state s:

$$tau(s,T) = \{t \in T \mid t = (s, \tau, s')\}$$

We define the function $sum^k_{CN,D}((s_i, s_j), T_i, T_j)$ as the sum of the best compatibility degree of forward neighbours of state s_i and those of state s_j:

$$sum^k_{CN,D}((s_i,s_j),T_i,T_j) = \begin{cases} \sum_{(s_i,l_i,s'_i)\in T_i} max_{(s_j,l_j,s'_j)\in T_j}(\textit{lab-comp}(l_i,l_j) * COMP^{k-1}_{CN,D}[s'_i, s'_j]) \\ \qquad\qquad if\ \|Fw(s_i,T_i)\| \neq \varnothing\ and\ \|(Fw(s_j,T_j)\| \neq \varnothing \\ 0 \qquad\qquad otherwise \end{cases}$$

We are now able to define the function $obs\text{-}comp^k_{CN,D}$ *wrt.* to *UR* and *UC* notions presented in Section 3.2.

Unspecified Receptions. For all global state (s_1, s_2): (i) $obs\text{-}comp^k_{UR,\leftrightarrow}$ returns 1 if and only if every outgoing emission in $E(s_1, T_1)$ ($E(s_2, T_2)$, respectively) perfectly matches an outgoing reception in $R(s_2, T_2)$ ($R(s_1, T_1)$, respectively) and all synchronisations on those emissions lead to compatible states; (ii) $obs\text{-}comp^k_{UR,\leftrightarrow}$ returns 0 if there is a deadlock; (iii) otherwise, $obs\text{-}comp^k_{UR,\leftrightarrow}$ measures the best compatibility of every outgoing emission in $E(s_1, T_1)$ with the outgoing receptions in $R(s_2, T_2)$, leading to the neighbouring states which have the highest compatibility degree, and vice-versa.

Definition 9 (Unspecified Receptions) *Given a global state (s_1, s_2), the observational compatibility is computed wrt. UR as follows:*

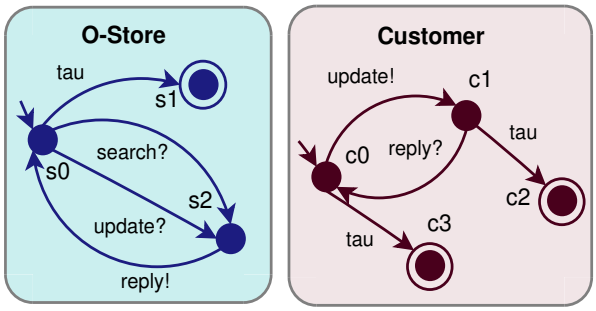


Figure 8: Online Store (I).

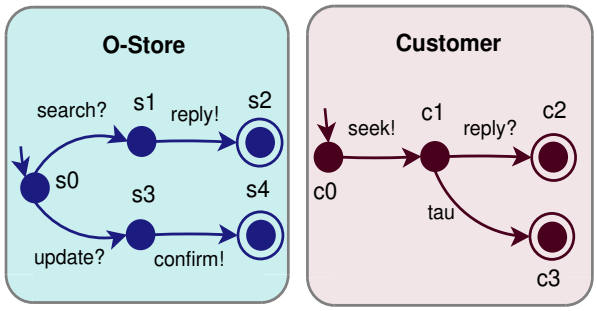


Figure 9: Online Store (II).

$$obs\text{-}comp^{k}_{UR,\leftrightarrow}((s_1,s_2)) = \begin{cases} 1 & \text{if } E(s_1,T_1) \cup E(s_2,T_2) = \varnothing \text{ and} \\ & ((s_1 \in F_1) \vee (tau(s_1,T_1) \neq \varnothing) \text{ or} \\ & (s_2 \in F_2) \vee (tau(s_2,T_2) \neq \varnothing)) \\ 0 & \text{if } E(s_1,T_1) \cup E(s_2,T_2) = \varnothing \text{ and} \\ & (s_1 \notin F_1) \wedge (tau(s_1,T_1) = \varnothing) \text{ and} \\ & (s_2 \notin F_2) \wedge (tau(s_2,T_2) = \varnothing) \\ \frac{sum^{k}_{UR,\leftrightarrow}((s_1,s_2),E(s_1,T_1),R(s_2,T_2)) + sum^{k}_{UR,\leftrightarrow}((s_2,s_1),E(s_2,T_2),R(s_1,T_1))}{\|E(s_1,T_1)\| + \|E(s_2,T_2)\|} & \\ & \text{otherwise} \end{cases}$$

Example 6 *In Figure 8, the* O-Store *service can receive either a search for an item or a cart update request, reply it, and end up in a final state after a time-out. The* Customer *first asks for its cart update and then waits for an answer. If no reply is received, the* Customer *can reach its final state after a time-out. Once the* Customer *receives a reply, he/she can either start another request or end up in a final state after execution of the τ action. Let us consider the global state* (s0, c0) *in Figure 8. Since both states* s0 *and* c0 *are not final and there is an emission going out from* c0*, $obs\text{-}comp^{1}_{UR,\leftrightarrow}((s_0,c_0))$ is computed using the third case in Definition 9. Here, there is a unique emission* update! *at* c0 *which perfectly matches with* update? *at* s0*, $lab\text{-}comp(update!, update?) = 1$. The synchronisation on these compatible labels leads to* (s2, c1) *where $COMP^{0}_{UR,\leftrightarrow}[\mathsf{s2},\mathsf{c1}] = 1$. Thus, at the first iteration:*

$$obs\text{-}comp^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c0})) = \frac{sum^{1}_{UR,\leftrightarrow}((\mathsf{c0},\mathsf{s0}),E(\mathsf{c0},T_{Client}),R(\mathsf{s0},T_{Server}))}{\|E(\mathsf{c0},T_{Client})\|}$$

where:

- $E(c_0,T_{Client}) = \{(c_0,update!,c_1)\}$
- $R(s_0,T_{Server}) = \{(s_0,update?,s_2),(s_0,search?,s_2)\}$
- $sum^{1}_{UR,\leftrightarrow}((c_0,s_0),E(c_0,T_{Client}),R(s_0,T_{Server})) = lab\text{-}comp(update!,update?) * COMP^{0}_{UR,\leftrightarrow}[\mathsf{s2},\mathsf{c1}] = 1.$

Hence, $obs\text{-}comp^{1}_{UR,\leftrightarrow}((s_0,c_0)) = 1$.

Unidirectional Complementarity. We assume that one state s_{er} (in the *complementer* protocol) perfectly complements the state s_{ed} (in the *complemented* protocol), *i.e.*, $obs\text{-}comp^{k}_{UC,\rightarrow}((s_{er},s_{ed})) = 1$, if there is a subset of outgoing observable transitions in $Fw(s_{er},T_{er})$ such that their respective labels are perfectly compatible with those of transitions in $Fw(s_{ed},T_{ed})$. Additionally, these transitions must lead into compatible states. If there is a deadlock, then this function returns 0. Otherwise, $obs\text{-}comp^{k}_{UC,\rightarrow}((s_{er},s_{ed}))$ measures the best compatibility of every transition label hold in $Fw(s_{er},T_{er})$ with those hold in $Fw(s_{ed},T_{ed})$, leading to the neighbouring states which have the highest compatibility degree.

Definition 10 (Unidirectional Complementarity) *Given a global state (s_{er}, s_{ed}), such that s_{er} and s_{ed} are states in the complementer and complemented service protocols, respectively, the observational compatibility is computed wrt. UC as follows:*

$$obs\text{-}comp^{k}_{UC,\rightarrow}((s_{er}, s_{ed})) = \begin{cases} 1 & \text{if } (sum^{k}_{UC,\rightarrow}((s_{ed}, s_{er}), T'_{ed}, T'_{er}) = \|Fw(s_{ed}, T_{ed})\|), \text{ or} \\ & ((s_{er} \in F_{er}) \wedge (s_{ed} \in F_{ed})), \text{ or} \\ & ((s_{ed} \in F_{ed}) \wedge (tau(s_{er}, T_{er}) \neq \varnothing)) \\ 0 & \text{if } ((s_{er} \notin F_{er} \vee s_{ed} \notin F_{ed})), \text{ and} \\ & (\bigcup(Fw(s_{er}, T_{er}), tau(s_{er}, T_{er}), \\ & \quad Fw(s_{ed}, T_{ed}), tau(s_{ed}, T_{ed})) = \varnothing \text{ or} \\ & \exists i, j \in \{ed, or\}, i \neq j, s_i \in F_i, s_j \notin F_j, Fw(s_j, T_j) \neq \varnothing, \\ & tau(s_j, T_j) = \varnothing) \\ \frac{sum^{k}_{UC,\rightarrow}((s_{ed}, s_{er}), T'_{ed}, T'_{er})}{max(\|T'_{ed}\|, \|T'_{er}\|)} & \text{otherwise} \end{cases}$$

where $T'_{ed} = Fw(s_{ed}, T_{ed})$ and $T'_{er} = Fw(s_{ed}, T_{er})$.

Example 7 *Let us consider the global state* $(\mathsf{s0}, \mathsf{c0})$ *in Figure 9. We want to check if the* O-Store *protocol complements the* Customer *protocol. Initially, there is only one observable message* seek! *in the* Customer *protocol which perfectly matches with message* search? *in the* O-Store *protocol (the message names are synonyms in Wordnet Similarity package),* $lab\text{-}comp(seek!, search?) = 1$. *In addition, the protocols can reach* $(\mathsf{s1}, \mathsf{c1})$ *through the synchronisation on these compatible labels such that* $COMP^{0}_{UC,\rightarrow}[\mathsf{s1}, \mathsf{c1}] = 1$. *Therefore, at the first iteration,* $obs\text{-}comp^{1}_{UC,\rightarrow}((\mathsf{s0}, \mathsf{c0})) = 1$ *because the condition of the first case in Definition 10 is satisfied as follows. Particularly,*

$$\begin{aligned} sum^{1}_{UC,\rightarrow}((\mathsf{c0}, \mathsf{s0}, T'_{\mathsf{Customer}}, T'_{\mathsf{O-Store}}) &= \|Fw(\mathsf{c0}, T_{\mathsf{Customer}})\| \\ &= 1 \end{aligned}$$

where

- $sum^{1}_{UC,\rightarrow}((\mathsf{s0}, \mathsf{c0})), T'_{\mathsf{O-Store}}, T'_{Costomer}) = lab\text{-}comp(seek!, search?) * COMP^{0}_{UC,\rightarrow}[\mathsf{s1}, \mathsf{c1}]$, *and*
- $Fw(\mathsf{c0}, T_{\mathsf{Customer}}) = \{\mathsf{seek!}\}$.

As far as τ transitions are concerned, we define the function $fw\text{-}propag^{k}_{CN,D}$, $D \in \{\leftrightarrow, \rightarrow\}$, which handles these internal behaviours based upon either a bidirectional or unidirectional compatibility propagation:
Bidirectional Compatibility. Regarding the bidirectional propagation, the compatibility is computed from both services point of view. The function $fw\text{-}propag^{k}_{CN,\leftrightarrow}((s_1, s_2))$ propagates to (s_1, s_2) the compatibility degrees obtained for the forward neighbours of state s_1 with those of state s_2 (using function $d\text{-}fw\text{-}propag^{k}_{CN,D}((s_1, s_2))$), and vice-versa (using function $d\text{-}fw\text{-}propag^{k}_{CN,D}((s_2, s_1))$). For each τ transition, $fw\text{-}propag^{k}_{CN,\leftrightarrow}$ must be checked on the target state. Observable transitions going out from (s_1, s_2) are compared using the function $obs\text{-}comp^{k}_{CN,\leftrightarrow}((s_1, s_2))$.

Definition 11 (Bidirectional Forward Propagation) *Given a global state* (s_1, s_1)*:*

$$fw\text{-}propag^{k}_{CN,\leftrightarrow}((s_1, s_2)) = \frac{d\text{-}fw\text{-}propag^{k}_{CN,\leftrightarrow}((s_1, s_2)) + d\text{-}fw\text{-}propag^{k}_{CN,\leftrightarrow}((s_2, s_1))}{2}$$

such that $\forall i, j \in \{1,2\},\ i \neq j$

$$d\text{-}fw\text{-}propag^{k}_{CN,\leftrightarrow}((s_i,s_j)) = \begin{cases} \frac{\sum_{(s_i,\tau,s'_i)\ \in T_i} fw\text{-}propag^{k}_{CN,\leftrightarrow}((s'_i,s_j))}{\|tau(s_i,T_i)\|} \\ \qquad\qquad \textit{if } tau(s_i,T_i) \neq \varnothing,\ \textit{and } \|Fw(s_i,T_i)\| = \varnothing \\ \frac{\sum_{(s_i,\tau,s'_i)\ \in T_i} fw\text{-}propag^{k}_{CN,\leftrightarrow}((s'_i,s_j)) + obs\text{-}comp^{k}_{CN,\leftrightarrow}((s_i,s_j))}{\|tau(s_i,T_i)\|+1} \\ \qquad\qquad \textit{otherwise} \end{cases}$$

Example 8 *Let us consider again the global state* $(\mathsf{s0},\mathsf{c0})$ *in Figure 8 and the UR notion. We show below the computation of* $fw\text{-}propag^{1}_{UR,\leftrightarrow}$ *at the initial global state, and which results in the average of the auxiliary values computed from each protocol point of view:*

$$fw\text{-}propag^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c0})) = \tfrac{1}{2} * (\tfrac{fw\text{-}propag^{1}_{UR,\leftrightarrow}((\mathsf{s1},\mathsf{c0})) + obs\text{-}comp^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c0}))}{2} + \tfrac{fw\text{-}propag^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c3})) + obs\text{-}comp^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c0}))}{2})$$

where:

- $fw\text{-}propag^{1}_{UR,\leftrightarrow}((\mathsf{s1},\mathsf{c0})) = obs\text{-}comp^{1}_{UR,\leftrightarrow}((\mathsf{s1},\mathsf{c0})) = 0$ *due to the deadlock that can occur at state* $(\mathsf{s1},\mathsf{c0})$.
- $fw\text{-}propag^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c3})) = obs\text{-}comp^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c3})) = 0$ *because there is also a deadlock at state* $(\mathsf{s0},\mathsf{c3})$.
- $obs\text{-}comp^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c0})) = lab\text{-}comp(update?, update!) * COMP^{0}_{UR,\leftrightarrow}[\mathsf{s2},\mathsf{c1}] = 1$.

As a consequence, $fw\text{-}propag^{1}_{UR,\leftrightarrow}((\mathsf{s0},\mathsf{c0})) = \frac{1}{2}$.

Unidirectional Compatibility. The function $fw\text{-}propag^{k}_{CN,\rightarrow}((s_1,s_2))$ is computed from STS_2 point of view, *i.e.*, the service compatibility is governed by STS_2's requirements. First, if there exists τ transitions at s_1, this state is considered compatible with s_2 if $fw\text{-}propag^{k}_{CN,\rightarrow}((s'_1,s_2)) = 1$ for every (s_1,τ,s'_1) (see first case in Definition 12). This check ensures that each time STS_1 traverses an internal transition, this protocol is able to fulfill the STS_2's requirements at the target state. If the last condition does not hold, we need to compute $fw\text{-}propag^{k}_{CN,\rightarrow}((s'_1,s_2))$ for every (s_1,τ,s'_1), and also the compatibility of observable transitions going out from (s_1,s_2) using $obs\text{-}comp^{k}_{CN,D}((s_1,s_2))$ (see second case in Definition 12). Otherwise, if no (s_1,τ,s'_1) exists, we compute $fw\text{-}propag^{k}_{CN,\rightarrow}((s_1,s'_2))$ for every (s_2,τ,s'_2), and also $obs\text{-}comp^{k}_{CN,D}((s_1,s_2))$ (see third case in Definition 12). Last, if no τ transition exists at (s_1,s_2), we deduce that $fw\text{-}propag^{k}_{CN,\rightarrow}((s_1,s_2)) = obs\text{-}comp^{k}_{CN,D}((s_1,s_2))$.

Definition 12 (Unidirectional Forward Propagation) *Given a global state* (s_1,s_2)*:*

$$fw\text{-}propag^{k}_{CN,\rightarrow}((s_1,s_2)) = \begin{cases} 1 \qquad \textit{if } tau(s_1,T_2) \neq \varnothing \textit{ and} \\ \qquad \sum_{(s_1,\tau,s'_1)\ \in T_1} fw\text{-}propag^{k}_{CN,\rightarrow}((s'_1,s_2)) = \|tau(s_1,T_1)\| \\ \frac{(\sum_{(s_1,\tau,s'_1)\ \in T_1} fw\text{-}propag^{k}_{CN,\rightarrow}((s'_1,s_2))) + obs\text{-}comp^{k}_{CN,\rightarrow}((s_1,s_2))}{\|tau(s_1,T_1)\|+1} \\ \qquad \textit{if } tau(s_1,T_1) \neq \varnothing \textit{ and} \\ \qquad \sum_{(s_1,\tau,s'_1)\ \in T_1} fw\text{-}propag^{k}_{CN,\rightarrow}((s'_1,s_2)) \neq \|tau(s_1,T_1)\| \\ \frac{(\sum_{(s_2,\tau,s'_2)\ \in T_2} fw\text{-}propag^{k}_{CN,\rightarrow}((s_1,s'_2))) + obs\text{-}comp^{k}_{CN,\rightarrow}((s_1,s_2))}{\|tau(s_2,T_2)\|+1} \\ \qquad \textit{otherwise} \end{cases}$$

Example 9 *Let us show the computation of* $fw\text{-}propag^{1}_{UC,\rightarrow}$ *at states* $(\mathsf{s0},\mathsf{c0})$ *and* $(\mathsf{s1},\mathsf{c1})$ *in Figure 9. First,* $fw\text{-}propag^{1}_{UC,\rightarrow}((\mathsf{s0},\mathsf{c0})) = obs\text{-}comp^{1}_{UC,\rightarrow}((\mathsf{s0},\mathsf{c0})) = 1$ *because no* τ *transition exists. Since there is one* τ *transition at* $\mathsf{c1}$*:*

$$fw\text{-}propag^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c1})) = \frac{fw\text{-}propag^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c3})) + obs\text{-}comp^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c1}))}{2}$$

	s0	s1	s2	s3	s4
c0	**0.78**	0.01	0.01	0.01	0.01
c1	0.01	0.68	0.01	0.35	0.01
c2	0.01	0.01	0.90	0.01	0.67
c3	0.01	0.45	0.76	0.35	0.76

Table 2: The compatibility matrix $COMP^{7}_{UC,\rightarrow}$.

where:

- $\mathit{fw\text{-}propag}^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c3})) = \mathit{obs\text{-}comp}^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c3})) = 0$ *due to the deadlock at state* $(\mathsf{s1},\mathsf{c3})$.
- $\mathit{obs\text{-}comp}^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c1})) = \mathit{lab\text{-}comp}(reply!, reply?) * COMP^{0}_{UC,\leftrightarrow}[\mathsf{s2},\mathsf{c2}] = 1$.

Hence, $\mathit{fw\text{-}propag}^{1}_{UC,\rightarrow}((\mathsf{s1},\mathsf{c1})) = \frac{1}{2}$.

State Compatibility. The function $\mathit{state\text{-}comp}^{k}_{CN,D}((s_1,s_2))$ computes the weighted average of three measures: the forward and backward compatibilities, and the value returned by the function $nat(s_1,s_2)$.

$$\mathit{state\text{-}comp}^{k}_{CN,D}((s_1,s_2)) = \frac{w_1 * \mathit{fw\text{-}propag}^{k}_{CN,D}((s_1,s_2)) + w_2 * \mathit{bw\text{-}propag}^{k}_{CN,D}((s_1,s_2)) + w_3 * nat(s_1,s_2)}{w_1 + w_2 + w_3}$$

where the weights w_1, w_2 and w_3 are automatically computed. w_1 (w_2, respectively) denotes the number of best matching found among the outgoing (incoming, respectively) transition labels in states s_i and s_j. w_3 is a binary weight which is set to 0 if there is at least one state with outgoing or incoming τ transitions, and such that both forward and backward compatibilities are equal to 1. Otherwise, w_3 is set to 1.

Compatibility Flooding. The compatibility degree of (s_1,s_2) at the k^{th} iteration is computed as the average of its previous compatibility degree at the $k-1^{th}$ iteration and the current state compatibility degree:

$$COMP^{k}_{CN,D}[s_1,s_2] = \frac{COMP^{k-1}_{CN,D}[s_1,s_2] + \mathit{state\text{-}comp}^{k}_{CN,D}((s_1,s_2))}{2}$$

Our iterative process terminates when the Euclidean difference $\varepsilon_k = \| COMP^{k}_{CN,D} - COMP^{k-1}_{CN,D} \|$ of matrices $COMP^{k}_{CN,D}$ and $COMP^{k-1}_{CN,D}$ converges.

Example 10 *Table 2 shows the matrix computed for the example depicted in Figure 9 according to the UC notion. This matrix was obtained after 7 iterations. Let us comment the compatibility of states* c0 *and* s0*. The measure is quite high because both states are initial and the emission* seek! *at* c0 *perfectly matches the reception* search? *at* s0*. However, the compatibility degree is less than 1 due to the backward propagation of the deadlock from the global state* $(\mathsf{s1},\mathsf{c3})$ *to* $(\mathsf{s1},\mathsf{c1})$*, and then from* $(\mathsf{s1},\mathsf{c1})$ *to* $(\mathsf{s0},\mathsf{c0})$.

Mismatch Detection. Our compatibility measure comes with a list of mismatches which identifies the incompatibility sources, *e.g.,* unmatched message names, different state natures or unshared parameter types. For instance, the states s0 and c1 in Figure 9 present several mismatches, *e.g.*, the first state is initial while the second is not, and their outgoing transition labels have the same directions.

Extensibility. Our approach is generic and can be easily extended to integrate other compatibility notions. Adding a compatibility notion CN only requires to define a new function $\mathit{obs\text{-}comp}^{k}_{CN,D}$.

4.3 Analysis of Compatibility Measures

In this section, we propose some techniques for automatically analysing the measures obtained from the compatibility matrix. We first present how the Boolean compatibility can be computed from the matrix. In the case of incompatible services, we propose some techniques for computing a global compatibility measure.

Compatible Protocols. Our flooding algorithm ensures that every time a mismatch is detected in a reachable global state, its effect will be propagated to the initial states. Hence, the forward and backward compatibility propagation between neighbouring states implies that protocols are compatible if and only if their initial states are also compatible, *i.e.*, $COMP^{k}_{CN,D}[I_1,I_2] = 1$.[5] Such information is useful for automatically discovering

[5] In this section, k stands for the last performed iteration.

available services that can be composed without using any adaptor service for compensating mismatches.

Global Protocol Compatibility. The global compatibility measure helps to differentiate between services that are slightly incompatible and those which are totally incompatible. This is useful to perform a first service selection step in order to find some candidates among a large number of services. Seeking for services with high global compatibility degree enables to simplify further processing to resolve their interface incompatibility, *e.g.*, using service adaptation [22].

The global compatibility can be computed differently depending on the user preferences. A first solution consists in computing the average of the maximal compatibility degrees computed for all states. Another alternative is to compute the global compatibility degree as the weighted average of all behavioural compatibility degrees that are higher than a threshold t. The weight is the rate of states having a compatibility degree higher than t, among all states compared in one service, with the states in the partner service. Algorithm 1 defines a function $global\text{-}comp$ which accepts as input two state sets S_1 and S_2, the matrix $COMP^k_{CN,D}$ and a threshold t. For each state $s_1 \in S_1$, the nested "for" loops (lines 2-13) sum up the compatibility degree for all global state $(s_1, s_j)_{s_j \in S_2}$ such that $COMP^k_{CN,D}[s_1, s_j] \geq t$. If there exists at least one state (s_1, s_j) where $COMP^k_{CN,D}[s_1, s_j] \geq t$ (line 10), then the number of checked states is incremented (line 11). Finally, the obtained sum is normalised, *i.e,* divided by the number of the performed sum operations (line 15), and multiplied with the weight $\frac{checked}{\|S_1\|}$.

Algorithm 1 $global\text{-}comp(S_1, S_2, COMP^k_{CN,D}, t)$

// computes the global compatibility degree from a matrix $COMP^k_{CN,D}$
inputs $S_1, S_2, t, COMP^k_{CN,D}$
output $global$

```
1: global := 0, count := 0, checked := 0
2: for all s1 ∈ S1 do
3:     match := False
4:     for all s2 ∈ S2 do
5:         if COMP^k_{CN,D}[s1, s2] ≥ t then
6:             global := global + COMP^k_{CN,D}[s1, s2]
7:             match := True; count = count + 1
8:         end if
9:     end for
10:    if match == True then
11:        checked = checked + 1
12:    end if
13: end for
14: if count ≠ 0 then
15:    global := global/count * checked/||S1||
16: end if
17: return global
```

Algorithm 1 computes the global compatibility measure from one STS point of view, and this works for the unidirectional compatibility notions. Regarding the bidirectional compatibility notions, the global compatibility is computed as the average of the values returned by functions $global\text{-}comp(S_1, S_2, COMP^k_{CN,D}, t)$ and $global\text{-}comp(S_2, S_1, COMP^k_{CN,D}, t)$.

Example 11 *Given a threshold* $t = 0.7$ *and the matrix in Table 2, running Algorithm 1 returns the following global compatibility degree* $global\text{-}comp(\mathsf{S_{O-Store}}, \mathsf{S_{Customer}}, COMP^7_{UC,\rightarrow}, 0.7) = 0.6$. *This measure is lower than 0.7 because* $\frac{checked}{\|\mathsf{S_{Customer}}\|} = \frac{3}{4}$, *i.e., the state* c2 *in the* Customer *has no match with any state in the* O-Store *such that* $\forall \mathsf{j} \in \mathsf{S_{O-Store}}, \nexists COMP^7_{UC,\rightarrow}[\mathsf{c2}, \mathsf{sj}] \geq 0.7$.

5 Prototype Tool and Experimental Results

Prototype Tool. Our approach for measuring the compatibility degree of service protocols has been fully implemented in a prototype tool called Comparator [26]. The framework architecture is given in Figure 10. The Comparator tool, implemented in Python, accepts as input two XML files corresponding to the service interfaces and an initial configuration, *i.e.*, the compatibility notion, the checking direction, and a threshold t. The tool returns the compatibility matrix, the mismatch list, and the global compatibility degree which indicates how compatible both services are. The implementation of our proposal is highly modular which makes easy its extension with new compatibility notions, or other strategies for comparing message names and parameters.

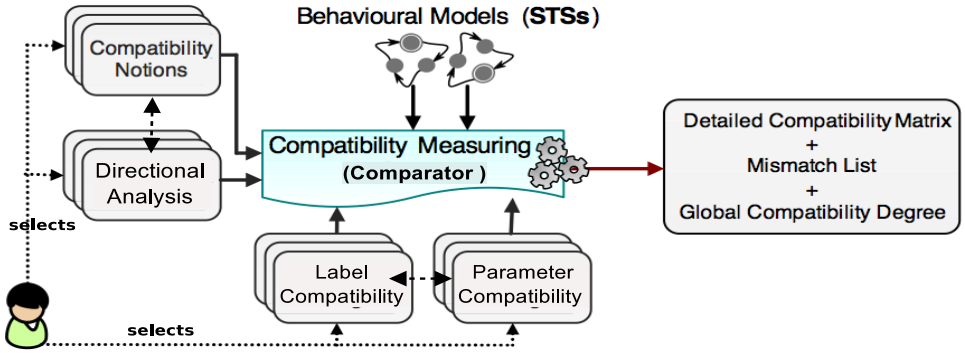


Figure 10: Comparator Architecture.

Experimental Results. So far, we have validated our prototype tool on more than 110 real-world examples, *e.g.*, a car rental, a travel booking system, a hard disk manager, a medical management system, an online email service. Table 3 summarises the experimental results of some of the examples of our database. Experimentations have been carried out on a Mac OS machine running on a 2.53 GHz Intel dual core processor with 4 GB of RAM. The computation time differs with respect to the input interface size (states and transitions). Experiments show that small examples with few states and transitions (*e.g.*, Ex9, Ex44, Ex71) require a negligible time for measuring their compatibility, whereas bigger examples (*e.g.*, Ex90, Ex101) need more time (see Table 3). The computation time increases with respect to the number of τ branchings and loops. For instance, Ex85 is quite big but consists of protocols with sequential structure and including very few loops, therefore the computation time does not exceed two minutes. On the other hand, protocols involving many loops (*e.g.*, Ex9) require more time (and iterations) than those having only few loops (*e.g.*, Ex85). To sum up, experiments have shown that Comparator computes the compatibility degree of quite large systems (*e.g.*, services with 210 states and 225 transitions) in a reasonable time (many iterations are performed in a few minutes). In addition, the returned compatibility measures were very satisfactory. As an example, each time a couple of states in two protocols presents several mismatches, this corresponds to a low value in the matrix and vice-versa. The reader may refer to [26] for some case studies illustrating the preciseness of our compatibility measure.

Evaluation. For the evaluation of our approach, the first author has been considered as an expert to validate the measure automatically computed *wrt.* the manual verification of the protocol mismatches. This study has shown that the prototype tool is well suited for checking the protocol compatibility. The returned measures were satisfactory, that is, each time a couple of states in two protocols presents several mismatches, this corresponds to a low value in the matrix and vice-versa.

The reader may refer to [26] for some case studies illustrating the preciseness of our compatibility measure. Although it can be relatively easy to compare some small protocols by hand, protocol verification is often recognised as a hard and tedious issue, specially in case of large and complex systems. In such a case, manual processing can be time-consuming, error-prone and hard if not possible to be achieved, specially if the user is not well familiarized with the protocol issues. The conducted experiments proved that our automatic analysis covers the aforementioned issues. The resulting matrix and detected mismatch list have always justified the difficulty of reusing existing services into new systems. The use of Comparator has allowed us to compute the compatibility degree of quite large systems (*e.g.*, services with 210 states and 225 transitions) in a reasonable time – many iterations were performed in a few minutes.

For the accuracy evaluation, we reuse the well-known precision and recall metrics [30] to estimate how much the measure automatically computed meets the expected result. Precision measures the matching quality (number of false positive matching) and is defined as the ratio of the number of correct state matching found to the total of state matching found. Recall is the coverage of the state matching results and is defined as the ratio of the the number of correct state matching found to the total of all correct state matching in two protocols. An effective measuring should produces a highest precision and recall, though these metrics are inversely related in the reality. In fact, the users can tolerate a small decrease in precision if it can bring a comparable increase of recall. The intuition behind this is that it would be easier to remove incorrect match than find missing ones.

We have studied the precision and recall for a set of significative examples from our database. We assume (s_i, s_j) is a best match if the state $s_i \in S_i$ has the highest compatibility degree with $s_j \in S_j$ among those in S_j. Our measuring process yields a precision and recall of 100% for compatible protocols, see for instance the

Example	States	Transitions	Compatibility Notion	$global$	Time (mn)	Iterations	ε
Ex9	8/5	8/5	UR	0.29	0m0.415s	8	0.01
			UC	0.18	0m10.581s	8	0.01
Ex44	20/22	19/21	UR	1	0m4.440s	8	0.02
			UC	0.81	0m13.860s	3	0.19
Ex71	20/4	19/3	UR	1	0m4.112s	8	0.02
			UC	1	0m5.848s	9	0.01
Ex85	59/59	64/75	UR	0.70	1m1.717s	4	0.13
			UC	0.69	0m47.513s	3	0.17
Ex90	86/86	90/90	UR	0.72	8m15.806s	7	0.04
			UC	0.74	2m48.400s	3	0.19
Ex101	124/86	135/90	UR	0.69	19m0.575s	10	0.02
			UC	0.70	8m0.460s	6	0.13

Table 3: Some Experimental Results ($t = 0.7$).

medical management case study presented in [26]. The evaluation has proved the effectiveness of our approach for matching incompatible protocols. For instance, the study of the car rental system [26] – which provides a service for car rental and an example of user requirements – produces a precision and recall equal to 85% and 95%, respectively. We applied the same evaluation to a flight advice system [26] which helps travelers to find flight information. This yields a precision and a recall equal to 91% and 100%, respectively. Other studies applied to the rest of examples have shown a good trade-off between precision and recall such that both often achieve high values.

6 Related Work

Analysing service protocols using quantitative techniques is emerging as an alternative to the classical approaches which usually return a Boolean result. Existing works which are devoted to behavioural analysis focus on two close research areas. The first one is known as substitutability checking and aims at finding correspondences between similar services. The second one is referred to as compatibility checking and verifies whether interacting services fulfill each other's requirements. Regarding related approaches, we focus on three kinds of techniques used for measuring the similarity or the compatibility of service interfaces.

Simple Protocol Traversal. The work given in [31] focused on two notions, namely, simulation and bisimulation to measure the similarity of computer viruses described using Labelled Transition Systems (LTSs). The similarity techniques use weighted quantitative functions which consist in a simple (not iterative), forward, and parallel traversal of two LTSs. The authors take inspiration from the existing equivalence relations to propose a strong and weak definition for their simulation and bisimulation notions, respectively. The weak relations allow one LTS to perform a special stuttering step ϵ, which is similar to a τ transition, whenever two LTSs cannot perform a transition with same labels. This work does not return the differences which distinguish one service from another, and there is no computation of a global similarity degree. Such a measure is useful for, *e.g.*, ranking and selecting services. In [32], the authors proposed a compatibility notion for checking two services described using the π-calculus. According to their proposal, two services are compatible if there is always at least one transition sequence between them, until reaching final states. This compatibility definition is too weak in the sense that when composing services together the deadlock-freeness cannot be guaranteed. Considering their compatibility definition, the authors present a method to compute the compatibility degree of two services as the average of the number of successful transition sequences. The compatibility computation relies on a simple and parallel traversal of protocols. This work does not compute the detailed compatibility of different protocol states, and also there is no mismatch detection.

Edit Distance. In [19, 1], the author assume that a service interface is initially synthesized to be compatible with the environment requirements, *i.e.*, referred to as correct interface. They claim that the interface can undergo changes which give rise to incompatibilities with its environment, *i.e.*, the service interface becomes defective. Thus, their proposal calculates the minimal edit distance between a given *defective* service and *synthesized correct* services. [19] extends the simulation algorithm given in [31] in order to correct deadlocking choreographies. In particular, it detects the modifications needed to achieve service simulation and make the choreography free of deadlocks. On the other hand, [1] focuses on the calculus of the differences between two versions of one service interface described using finite State Machines (FSM). The quantitative simulation measures the state similarity

based on the analysis of outgoing transition labels without any semantic comparison of these label names. This measuring technique does not consider the similarity of neighbouring states, therefore the main advantage of propagation-based approach is missing. This approach computes a global similarity measure.

Similarity Flooding. In [23, 25], the similarity flooding technique was applied to the problem of model matching. This algorithm returns a matrix for the similarity propagation which is updated iteratively by fixpoint computation. In [23], Melnik *et al.* propose a set of metrics to measure correspondences between elements of data structures such as data schemas, data instances or a combination of both, described with directed labelled graphs. Their approach considers a forward and backward similarity propagation. This work aims at assisting human developers in matching elements of a schema by suggesting candidates. However, the implemented tool does not enable a fully automated matching because the user can manually adjust (delete and/or add) some matches. The *match* operator introduced in [25] measures the similarity (correspondences) between models which can represent different versions or variants across a family of software units described using Statecharts. The similarity measuring combines a set of static and behavioural matchings. The behavioural matching is computed using a flooding algorithm and relies on a bisimulation notion presented in [31]. In this work, the behavioural similarity is computed as the maximum of forward and backward behavioural matching. By doing so, it is not possible to detect the Boolean similarity from the initial states. More recently, [16] propose a semi-automated approach for checking the matching of messages in two business process models such that the computed values can be updated depending on the user feedback. The authors combine a depth and flooding-based interface matching for measuring the behavioural compatibility of two interacting protocols. This work aims at detecting the message merge/split mismatch in order to help the automatic specification of adaptation contacts. An interesting state-of-the-art on measuring the similarity of business process models is given in [27].

Although the approaches presented so far consider different techniques to reason on service behaviours, a very little attention was paid to value-passing, internal behaviours, and the semantical comparison of message names. Considering such features enables to avoid several interaction issues. Our approach is also different since we focus on measuring the compatibility of process-oriented models which are understood as refinements of business process models [11]. Most of existing techniques rely on in a simple forward traversal of protocols, and also lack the detection of mismatches or correspondences. The mismatch detection is of utmost importance to fix and compensate the interaction issues. Another weakness of existing approaches is that the detailed comparison of service states can be less useful for ranking and selecting services, and then computing a global measure from the detailed ones is a good alternative. Our approach overcomes all these limits. Moreover, quantifying service behaviours is commonly used for similarity measuring, whereas we focus on compatibility measuring. We summarise in Table 4[6] the comparison of existing work with our proposal.

		[25]	[31]	[19]	[1]	[32]	[16]	Our approach
Model	Messages and protocols	√	√	√	√	√	√	√
	Value-passing	×	×	×	×	×	×	√
	Internal actions	×	×	×	×	√	×	√
	Description language	Statechart	LTS	Finite Automaton	FSM	π-calculus	FSM	STS
Analysis	Issue	Similarity	Similarity	Similarity	Similarity	Compatibility	Compatibility	Compatibility
	Notion(s)	BIS	(WK/ST) SIM/BIS	WK SIM	SIM	OP	DF	UR/UC/…
Computation	Message semantics	√	×	×	×	×	√	√
	Processing	Iterative	Simple	Simple	Simple	Simple	Iterative	Iterative
	Technique	Flooding	Parallel traversal	Edit distance	Edit distance	Parallel traversal	Flooding	Flooding
	Detailed measures	√	√	√	√	×	√	√
	Mismatch detection	×	×	√	√	×	√	√
	Global measure	×	×	×	√	√	×	√
Tool support		√	√	√	√	√	√	√

Table 4: A Summary of Approaches Based on Quantitative Behavioural Analysis.

7 Conclusion

To the best of our knowledge, we are the first who suggest a generic framework to automatically measure the compatibility degree of service interfaces. Our measuring method relies on a compatibility flooding algorithm, and is parameterised by different compatibility notions. In addition to computing the matrix and the global measure of compatibility, a list of mismatches is returned. Our proposal is fully supported by the Comparator tool

[6] BIS, SIM, OP, DF, UR, UC, WK, and ST are used as abbreviations of bisimulation, simulation, one path, deadlock-freedom, unspecified receptions, unidirectional complementarity, weak, and strong, respectively.

which has been validated on many examples. This work has some straightforward applications in the software adaptation area. Our tool was already integrated into an environment for the interactive specification of adaptation contracts [6]. Our main perspective is to apply our compatibility measuring approach for the automatic generation of adaptor protocols.

Acknowledgements. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN) and FEDER, and by the project P06-TIC220, funded by the Andalusian government.

References

[1] A. Aït-Bachir. Measuring Similarity of Service Interfaces. In *Proc. of the PhD Symposium at ICSOC'08*, volume 421 of *CEUR Workshop Proceedings*, 2008.

[2] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of TES'04*, volume 3324 of *LNCS*, pages 15–28. Springer, 2004.

[3] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[4] M. Bravetti and G. Zavattaro. Contract-Based Discovery and Composition of Web Services. In *SFM'09*, volume 5569 of *LNCS*, pages 261–295. Springer, 2009.

[5] J. Cámara, J. Antonio Martín, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In *Proc. of ICSE'09*, pages 627–630. IEEE, 2009.

[6] J. Cámara, G. Salaün, C. Canal, and M. Ouederni. Interactive Specification and Verification of Behavioural Adaptation Contracts. In *Proc. of QSIC'09*, pages 65–75. IEEE Computer Society, 2009.

[7] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and Inheritance in Software Architectures. *Sci. Comput. Program.*, 41(2):105–138, 2001.

[8] R. Cleaveland and O. Sokolsky. Equivalence and Preorder Checking for Finite-State Systems. *Handbook of Process Algebra*, pages 391–424, 2001.

[9] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat. A Model-Based Approach to the Verification and Adaptation of WF/.NET Components. In *Proc. of FACS'07*, volume 215 of *ENTCS*, pages 39–55, 2008.

[10] L. de Alfaro and T. Henzinger. Interface Automata. In *Proc. of ESEC/FSE'01*, pages 109–120. ACM Press, 2001.

[11] R. M. Dijkman, M. Dumas, and L. García-Bañuelos. Graph Matching Algorithms for Business Process Model Similarity Search. In *Proc. of BPM'09*, volume 5701 of *LNCS*, pages 48–63. Springer, 2009.

[12] F. Durán, M. Ouederni, and G. Salaün. Checking Protocol Compatibility using Maude. In *Proc. of FOCLASA'09*, volume 255, pages 65–81. ENTCS, 2009.

[13] H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of ICSE'06*, pages 771–774. ACM Press, 2006.

[14] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of WWW'04*, pages 621–630. ACM Press, 2004.

[15] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.

[16] G. Y. Xu H. R. M. Nezhad and B. Benatallah. Protocol-aware Matching of Web Service Interfaces for Adapter Development. In *Proc. of WWW'10*, pages 731–740. ACM, 2010.

[17] N. Hameurlain. Flexible Behavioural Compatibility and Substitutability for Component Protocols: A Formal Specification. In *Proc. of SEFM'07*, pages 391–400. IEEE Computer Society, 2007.

[18] M. Hennessy and H. Lin. Symbolic Bisimulations. *TCS*, 138(2):353–389, 1995.

[19] N. Lohmann. Correcting Deadlocking Service Choreographies Using a Simulation-Based Graph Edit Distance. In *Proc. of BPM'08*, volume 5240 of *LNCS*, pages 132–147. Springer, 2008.

[20] C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[21] J. A. Martín and E. Pimentel. Automatic Generation of Adaptation Contracts. In *Proc. of FOCLASA'08*, volume 229 of *ENTCS*, pages 115–131. Elsevier, 2009.

[22] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In *Proc. of ICSOC'08*, volume 5364 of *LNCS*, pages 84–99. Springer, 2008.

[23] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proc. of ICDE'02*, pages 117–128. IEEE Computer Society, 2002.

[24] R. Milner, J. Parrow, and D. Walker. Modal Logics for Mobile Processes. *TCS*, 114(1):149–171, 1993.

[25] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proc. of ICSE'07*, pages 54–64. ACM Press, 2007.

[26] M. Ouederni. Comparator Web Page. Available at `http://www.lcc.uma.es/~meriem/comparator.html`.

[27] C. Ouyang, M. Dumas, W. M. P. van der Aalst, A. H. M. ter Hofstede, and J. Mendling. From Business Process Models to Process-Oriented Software systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1), 2009.

[28] T. Pedersen, S. Patwardhan, and J. Michelizzi. WordNet::Similarity - Measuring the Relatedness of Concepts. In *Proc. of AAAI'04*, pages 1024–1025. AAAI, 2004.

[29] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *IJBPIM*, 1(2):116–128, 2006.

[30] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill New York, 1983.

[31] O. Sokolsky, S. Kannan, and I. Lee. Simulation-Based Graph Similarity. In *Proc. of TACAS'06*, volume 3920 of *LNCS*, pages 426–440. Springer, 2006.

[32] Z. Wu, S. Deng, Y. Li, and J. Wu. Computing Compatibility in Dynamic Service Composition. *Knowledge and Information Systems*, 19(1):107–129, 2009.

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