HAL
open science

Flooding-Based Algorithm for Behavioural Compatibility Measuring

Meriem Ouederni, Uli Fahrenberg, Axel Legay, Gwen Salaün

To cite this version:

Meriem Ouederni, Uli Fahrenberg, Axel Legay, Gwen Salaün. Flooding-Based Algorithm for Behavioural Compatibility Measuring. [Research Report] Inria Rennes. 2014. hal-01088157

HAL Id: hal-01088157

https://inria.hal.science/hal-01088157v1

Submitted on 27 Nov 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Flooding-Based Algorithm for Behavioural Compatibility Measuring

Meriem Ouederni[1], Uli Fahrenberg[2], Axel Legay[2], and Gwen Salaün[3]

[1] Toulouse INP/IRIT, France
[2] Irisa / Inria Rennes, France
[3] Grenoble INP/INRIA/LIG/CNRS, France

Abstract. Nowadays, large software systems are mostly built using existing services. These are not always designed to interact, i.e., their public interfaces often present some mismatches. Checking compatibility of service interfaces allows one to avoid erroneous executions when composing the services and ensures correct reuse and interaction. Service compatibility has been intensively studied, in particular for discovery purposes, but most of existing approaches return a Boolean result. In this paper, we present a quantitative approach for measuring the compatibility degree of service interfaces. Our method is generic and flooding-based, and fully automated by a prototype tool.

1 Introduction

In service oriented computing, software systems are mostly built using existing services. Services are considered as black boxes accessed through their public interfaces which present four interoperability levels [2], *i.e.*, signature, interaction protocols, quality of services, and semantics. These interfaces must be compatible in order to ensure the correct composition and reuse of loosely-coupled services. This paper deals with the compatibility verification of service interfaces and focuses on the interaction protocol level. Checking the compatibility of interaction protocols is a tedious and hard task, even though it is of utmost importance to avoid run-time errors, *e.g.*, deadlock situations or unmatched messages.

Most of the existing approaches (see [8] for a detailed survey) return a "True" or "False" result to detect whether services are compatible or not. Unfortunately, a Boolean answer is not very helpful for many issues. Firstly, 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. Secondly, a Boolean result does not give any detailed information on which parts of service protocols are compatible or not. Thirdly, regarding the incompatible parts of protocols, such a result typically does not come with a mismatch list which enables us to understand and then resolve the incompatibility issues.

To overcome the aforementioned limits, a new solution aims at *quantifying* the compatibility degree of service interfaces. This issue has been addressed by a few recent works, see for instance Related Work Section. However, most of them are based upon description models of service interfaces, *e.g.*, business

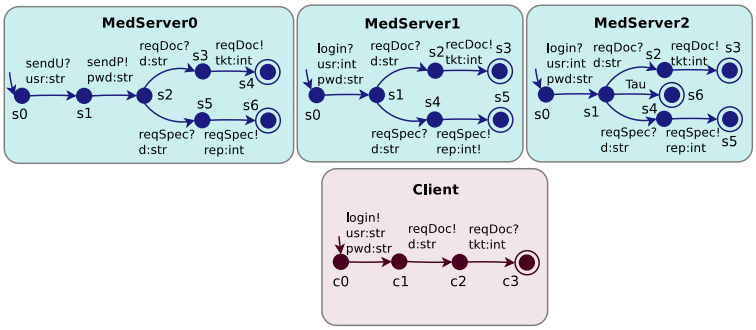


Fig. 1. A simple medical management system.

protocols [16], 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 [17]. Moreover, existing approaches, such as [27], measure the interface compatibility using a simple (*i.e.*, not iterative) traversal of protocols, and the results lack the preciseness which is essential for detecting subtle protocol mismatches. Lastly, a unique compatibility notion is always considered to check the services, and this makes the approaches useful only for specific application areas, *e.g.*, service choreography [11] or service adaptation [16].

As an example, we refer to the symbolic transition systems[4] in Fig. 1, describing an on-line medical management system which handles patient appointments within a health care institution, either with general practitioners or with specialist doctors. The Client can first log on to a server by sending her user name and password (login!). Then, she asks for an appointment with a general practitioner (reqDoc!) and receives an appointment identifier. We present three services for the medical server, which all seem to match the Client service, yet they all fail in subtle ways: MedServer0 can only receive user name and password separately, whereas Client wants to send them together; MedServer1 has a type mismatch on the usr parameter; and with MedServer2, communication can deadlock (if it silently proceeds to state s6, for example after a timeout). Hence none of the MedServer services are compatible with Client. We shall, however, later see that, using our quantitative techniques, there is a clear preference for MedServer1 in which the incompatibilities are much easier mended than in the other two.

In this paper, we propose a novel approach for quantifying the compatibility degree of interacting services. Instead of a Boolean result, we compute a numerical measure in the interval [0..1], where 0 means no compatibility and 1 means perfect compatibility. We describe service interfaces using a formal model, taking into account interaction protocols, *i.e.*, messages and their application order, but also value-passing and internal actions. We propose a generic framework where the compatibility degree of service interfaces can be automatically measured according

[4] We shall introduce symbolic transition systems more formally in Section 2.

to different compatibility notions. We illustrate our approach using bidirectional and unidirectional compatibility notions, namely *unspecified receptions* [28] and *unidirectional complementarity*; additional notions can easily be added to our framework. The compatibility is computed in two steps. First, we compute a set of static compatibility degrees where the execution order of messages is not taken into account. Then, we user a flooding algorithm to compute the compatibility degree of interaction protocols using the static compatibility results. The computation process also returns the mismatch list indicating the interoperability issues, and a global compatibility degree for two interaction protocols. Our solution is fully automated by a prototype tool Comparator [6] we have implemented.

This paper improves previous preliminary approaches [18,20] as follows. We give the mathematical definitions of our heuristics used for computing the compatibility of two service interfaces. We also show how our approach can be used for systems interacting using an asynchronous communication model. We prove, using Banach's fixed point theorem, that the flooding-based computation always converges. Finally, we present several experiments to better evaluate our prototype tool.

Quantifying protocol compatibility brings more advantages than the Boolean approaches, because it returns a detailed measure but also a list of mismatches that can be useful for many service applications, such as automatic service ranking, service discovery, composition, or adaptation. In the case of service adaptation [13] for instance, if a set of services are incompatible, the detailed measure and the mismatch list help to understand which 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.

2 Service Model and Notations

We describe service interfaces using interaction protocols represented by *Symbolic Transition Systems* (STSs). Our STSs are a variant of STGs (Symbolic Transition Graphs) [10] where guards are replaced with internal τ transitions. These transitions keep an abstraction closer to the service implementation and ensure (if services are compatible) a correct interaction whatever values are exchanged.

Definition 1. *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 $T \subseteq S \setminus F \times A \times S$ is the transition relation.*

A *label* is either the (internal) τ action or a tuple (m, d, pl) where m is the message name, d is 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, or a list of typed variables if the label is a reception. Here, services interact with each other based on a synchronous and binary communication model. The operational semantics of this model is formalised in [8].

The STS 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 programming languages (*e.g.*, Abstract BPEL – Business Process Execution Language or WF – Workflow) [3] for verification,

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="MedicalDatabase"/>
   <partnerLinks>
      <partnerLink name="DB" partnerLinkType="tns:MedDb"
      myRole="MedDbPortTypeRole"/>
   </partnerLinks>
   ...
   <sequence>
      <receive name="availability_REC_1" partnerLink="DB"
         operation="availability"
         portType="tns:MedDbPortType">
      </receive>
      ...
      <reply name="availability_INV_1" partnerLink="DB"
         operation="availability"
         portType="tns:MedDbPortType">
      </reply>
   </sequence>
</process>
```

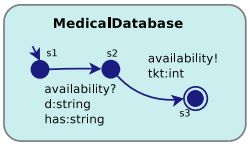


Fig. 2. Simplified BPEL and its STSs.

composition or adaptation purposes. For instance, BPEL is used by the service community for building large applications by reusing distributed and interacting services to reply complex user requests. Fig. 2 illustrates a simplified version of a BPEL example and the extracted STS.

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 in STS labels. We suppose that there are no cycles of internal transitions, *i.e.*, no transition sequences $(s_1, \tau, s_2), \ldots, (s_n, \tau, s_1)$.

3 Service Compatibility

Service compatibility is achieved if services can correctly interact with each other, *i.e.*, synchronisations over observable actions which are exchanged between services. Checking for correct service interaction needs to verify if service protocols satisfy a criterion, *i.e.*, compatibility notion. In this article, we compute the compatibility for two services and we distinguish two classes of notions, *i.e.*, bidirectional and unidirectional, depending on the direction of the compatibility checking. We particularly illustrate our approach with a bidirectional notion, namely *unspecified receptions* (*UR* for short), and with an unidirectional notion, namely *unidirectional complementarity* (*UC* for short). In the case of unidirectional compatibility checking, one of the two services plays a particular role because its *requirements* (messages) must all be satisfied by the partner service. This class can be useful for checking, *e.g.*, a client/server model. In this setting, a server service must be able to receive and answer all requests from a client, but this server can also handle other requests from other clients.

Before defining both *UR* and *UC*, we give below some preliminary concepts on which those compatibility notions rely:

Static Compatibility. *Parameter compatibility* requires that the parameter list expected to be received matches (same types in the same order) the parameter list coming with the sent message. *Label compatibility* requires labels to have opposite directions, same names, and compatible parameters.

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.

Deadlock-Freeness. This is required for checking successful system termination, *i.e.*, the services can always interoperate starting from a given global state until reaching final states. All the traversed global states belong to the set of deadlock-free states (referred to as *DF*) .

State Compatibility. Service interaction depends on synchronisation over observable actions and is defined using a criterion to be checked on reachable global states. For a given global state (s_1, s_2) of two protocols $STS_i = (A_i, S_i, I_i, F_i, T_i)$, this state is compatible if every message l_1 sent (received) by STS_1 at state s_1 will be eventually received (sent, respectively) by STS_2 at state s_2, such that both protocols evolve into a compatible global state, and vice-versa. If STS_2 is not able to interact with STS_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, both states (s_1, s'_2) and (s'_1, s''_2) must also be compatible. Note that we handle τ actions similarly to branching equivalence [26].

Additionally to the forward exploration above, state compatibility is determined by backtracking along transitions. Hence, every transition in one STS leading to (s_1, s_2) must much with a transition in the other STS where their labels are compatible. Furthermore both transitions must come from compatible states such that τ actions are handled similarly as stated above.

Unspecified Receptions (*UR*). This notion is inspired from [28] and requires that two services are compatible if (i) they are deadlock-free, 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. 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.

Unidirectional Complementarity (*UC*). Two services are compatible wrt. the *UC* notion if (1) they are deadlock-free and (2) one of them (the *complementer*) must eventually receive and send all messages that its partner (*complemented*) expects to send and receive, respectively, at all global reachable states. Hence, the *complementer* service may send and receive more messages than the *complemented* service. This asymmetric notion is useful to check the successful communication in the client/server model where a server can interact with clients with different behaviours. In this setting, each client behaviour must be satisfied (complemented) by the server.

4 Quantifying Compatibility

This section presents our techniques for measuring the compatibility of two service protocols. These techniques rely on the compatibility definitions given in Section 3. We compute the compatibility at several levels of service interfaces such as states,

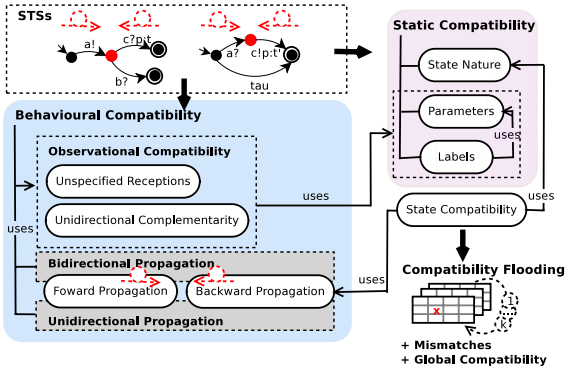


Fig. 3. Compatibility measuring process.

labels, and parameters. We aim at using all information described in service interfaces in order to get the highest precision for the computed compatibility. The final compatibility degree of two interaction protocols is computed relying on all these sources of compatibility, and following a *divide-and-conquer* approach.

For purpose of clarity, we assume in the rest of this article that the different functions defined have access to the $STS_i = (A_i, S_i, I_i, F_i, T_i)$ even if they are not explicitly passed as input parameters. However, we make parameters explicit if they are modified. The approach overviewed in Fig. 3 consists first in computing three static compatibility measures (Section 4.1) where the order of exchanged messages is not considered. In a second step, these static measures are used for computing the behavioural compatibility degree for all global states (Section 4.2). Lastly, we show how the global compatibility degree is computed (Section 4.3).

4.1 Static Compatibility

State Nature. We compare state nature using the function $nat((s_1, s_2))$. It returns 1 if states s_1 and s_2 have the same nature, *i.e.*, both are either initial, final or none of the two. Otherwise, $nat((s_1, s_2)) = 0$ returns 0:

Parameters. The compatibility degree of two lists of parameters exchanged with messages pl_1 and pl_2 depends on three auxiliary measures, namely: (i) the compatibility of parameter *number*, (ii) the compatibility of parameter *order*, and (iii) the compatibility of parameter *type*. These measures must be set to 1 if $pl_1 \cup pl_2 = \varnothing$. Otherwise, they are computed as follows:

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

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

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

Here, the function $unorderedTypes$ returns the set of parameter types existing in pl_1 and pl_2—*i.e.*, shared types—but which are not in the same order in both lists. The function $unsharedTypes$ returns the set of parameter types existing in only one list.

The function *par-comp* then computes the parameter compatibility as the average of the measures returned by the three previous functions:

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

Labels. We measure label compatibility as follows. Given a pair of labels $(l_1, l_2) \in A_1 \times A_2$ with $l_i = (m_i, d_i, pl_i)$, the function *lab-comp*(l_1, l_2) returns 0 if l_1 and l_2 have the same direction, and otherwise computes the average of the semantic compatibility of message names[5] and *par-comp*(pl_1, pl_2):

$$\textit{lab-comp}(l_1, l_2) = \begin{cases} 0 & \text{if } d_1 = d_2 \\ \frac{\textit{sem-comp}(m_1,m_2)+\textit{par-comp}(pl_1,pl_2)}{2} & \text{otherwise} \end{cases}$$

4.2 Behavioural Compatibility

We now present our metrics to compute the behavioural compatibility for two service protocols, $STS_i = (A_i, S_i, I_i, F_i, T_i)$, using the static measures previously introduced in Section 4.1. The intuition underlying theses metrics relies on the compatibility definitions given in Section 3.

We describe 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. Such a propagation relies on the intuition that two states are compatible if their backward and forward neighbouring states are compatible.

The flooding algorithm returns a matrix $CM^k_{CN,D}$. Each entry $CM^k_{CN,D}[s_1, s_2]$, stands for the compatibility measure of global state (s_1, s_2) at the kth iteration. The parameter *CN* refers to the considered compatibility notion, which is checked using either an unidirectional ($D = \rightarrow$) or a bidirectional ($D = \leftrightarrow$) protocol traversal. $CM^0_{CN,D}$ represents the initial compatibility matrix where all states are supposed to be perfectly compatible, *i.e.*, $\forall (s_1, s_2) \in S_1 \times S_2,\ CM^0_{CN,D}[s_1, s_2] = 1$.

In order to compute $CM^k_{CN,D}[s_1, s_2]$, we define two functions, *obs-comp*$^k_{CN,D}$ and *state-comp*$^k_{CN,D}$ detailed as follows. The first function, *observational compatibility*, computes the compatibility of outgoing and incoming observable transitions. The second function, *state compatibility*, propagates the compatibility from the forward and backward neighbouring states to (s_1, s_2), taking into account τ transitions and observational compatibility. The compatibility propagation is also parametrised according to the parameter D. In this article, we only present the forward compatibility, as the backward compatibility is handled in a similar way.

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

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

[5] We assume that message names match if they are synonyms according to the Wordnet similarity package [21].

We let $tau(s,T) = \{t \in T \mid t = (s,\tau,s')\}$ denote the set of τ-transitions emanating from a state s. We define the function $sum^k_{CN,D}((s_1,s_2),T_1,T_2)$ as the sum of the best compatibility degree of forward neighbours of state s_1 and those of state s_2:

$$sum^k_{CN,D}((s_1,s_2),T_1,T_2) = \begin{cases} \sum\limits_{(s_1,l_1,s_1')\in T_1} \max\limits_{(s_2,l_2,s_2')\in T_2} (lab\text{-}comp(l_1,l_2) \cdot CM^{k-1}_{CN,D}[s_1',s_2']) & \\ & \text{if } |Fw(s_1,T_1)| \neq 0 \text{ and } |(Fw(s_2,T_2)| \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

We are now able to define the function $obs\text{-}comp^k_{CN,D}$ according to the *UR* and *UC* notions presented in Section 3.

Unspecified Receptions. For a global state (s_1,s_2), $obs\text{-}comp^k_{UR,\leftrightarrow}$ returns 1 if there are no emissions from the states and they are deadlock free, and otherwise recursively measures the best compatibility of emissions with receptions, taking the compatibility of the states reached into account:

Definition 2. *Given a global state (s_1,s_2), the observational compatibility is computed wrt. the UR compatibility notion as follows:*

$$obs\text{-}comp^k_{UR,\leftrightarrow}((s_1,s_2)) = \begin{cases} 1 & \textit{if } E(s_1,T_1) \cup E(s_2,T_2) = \varnothing \textit{ and } (s_1,s_2) \in DF \\ 0 & \textit{if } (s_1,s_2) \notin DF \\ \frac{1}{|E(s_1,T_1)|+|E(s_2,T_2)|} \cdot \Big(sum^k_{UR,\leftrightarrow}((s_1,s_2),E(s_1,T_1),R(s_2,T_2)) & \\ \quad + sum^k_{UR,\leftrightarrow}((s_2,s_1),E(s_2,T_2),R(s_1,T_1)) \Big) & \textit{otherwise} \end{cases}$$

Unidirectional Complementarity. We compute how well one state s_{er} (in the *complementer* protocol) complements the state s_{ed} (in the *complemented* protocol). The comparison returns 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 in $Fw(s_{er},T_{er})$ with those in $Fw(s_{ed},T_{ed})$, leading to the neighbouring states which have the highest compatibility degree:

Definition 3. *For a global state (s_{er},s_{ed}), the observational compatibility is computed wrt. the UC compatibility notion as follows, for $T'_{ed} = Fw(s_{ed},T_{ed})$ and $T'_{er} = Fw(s_{er},T_{er})$:*

$$obs\text{-}comp^k_{UC,\rightarrow}((s_{er},s_{ed})) = \begin{cases} 1 & \textit{if } sum^k_{UC,\rightarrow}((s_{ed},s_{er}),T'_{ed},T'_{er}) = |Fw(s_{ed},T_{ed})| \textit{ and } (s_{ed},s_{er}) \in DF \\ 0 & \textit{if } (s_{ed},s_{er}) \notin DF \\ \frac{sum^k_{UC,\rightarrow}((s_{ed},s_{er}),T'_{ed},T'_{er})}{max(|T'_{ed}|,|T'_{er}|)} & \textit{otherwise} \end{cases}$$

As far as τ transitions are concerned, we define the function $\textit{fw-propag}^k_{CN,D}$, $D \in \{\leftrightarrow, \rightarrow\}$, which handles these internal behaviours based upon either a bidirectional or unidirectional compatibility propagation:

Bidirectional Compatibility. Here, compatibility is computed from both services' point of view. That is, for a given global state (s_1, s_2), we compute the compatibility of the forward neighbours of s_1 with those of s_2 and vice-versa. For each τ transition, $\textit{fw-propag}^k_{CN,\leftrightarrow}$ must be checked on the target state, and observable transitions out of (s_1, s_2) are compared using $\textit{obs-comp}^k_{CN,\leftrightarrow}$:

Definition 4. *Given a global state* (s_1, s_1)*:*

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

$$\textit{d-fw-propag}^k_{CN,\leftrightarrow}((s_1, s_2)) = \begin{cases} \frac{\sum_{(s_1,\tau,s'_1)\in T_1} \textit{fw-propag}^k_{CN,\leftrightarrow}((s'_1,s_2))}{|\textit{tau}(s_1,T_1)|} & \textit{if } \textit{tau}(s_1, T_1) \neq 0 \textit{ and } |Fw(s_1, T_1)| = 0 \\ \frac{\sum_{(s_1,\tau,s'_1)\in T_1} \textit{fw-propag}^k_{CN,\leftrightarrow}((s'_1,s_2)) + \textit{obs-comp}^k_{CN,\leftrightarrow}((s_1,s_2))}{|\textit{tau}(s_1,T_1)|+1} & \textit{otherwise} \end{cases}$$

Unidirectional Compatibility. To compute $\textit{fw-propag}^k_{CN,\rightarrow}((s_1, s_2))$, from the point of view of s_2 as the complemented state, we first follow any τ-transitions from s_1, and if there are no such transitions,[6] then we follow any τ-transitions from s_2. Measuring the compatibility after every τ-transition enables us to check whether this protocol is able to fulfil its partner requirements at the target state:

Definition 5. *Given a global state* (s_1, s_2)*:*

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

State Compatibility. We compute the weighted average of three measures, forward and backward compatibility and state nature:

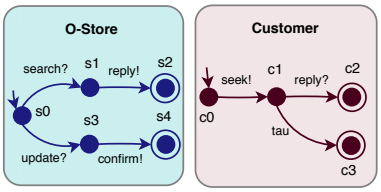
$$\textit{state-comp}^k_{UC,\rightarrow}(s_1, s_2) = \frac{w_1 \cdot \textit{fw-propag}^k_{UC,\rightarrow}(s_1,s_2) + w_2 \cdot \textit{bw-propag}^k_{UC,\rightarrow}(s_1,s_2) + \textit{nat}(s_1,s_2)}{w_1+w_2+1}$$

where the weights w_1 and w_2 denote the number of best matches found among the outgoing and incoming, respectively, transition labels in states s_1 and s_2.

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

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

[6] Given that we have excluded τ-loops, this will eventually be the case.



	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

Fig. 4. STSs of online store system (left) and associated compatibility matrix.

Example 6. The table in Fig. 4 (right) shows the matrix obtained, after 7 iterations, for the example depicted according to the *UC* notion. Let us comment on 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 (they are WordNet synonyms). However, the compatibility degree is less than 1 due to the backward propagation of the deadlock from the global state (s1, c3) to (s1, c1), and then from (s1, c1) to (s0, c0).

Convergence. In appendix, we give a detailed formal proof that the computation described here always converges to a unique compatibility matrix $CM_{CN,D}$. The argument is based on the fact, proven in the supplement, that the function on matrices defined in this section is a *contraction*, hence by Banach's fixed point theorem, it converges to a unique fixed point. For practical purposes, our iterative process is terminated when the Euclidean metric $\varepsilon_k = |CM^k_{CN,D} - CM^{k-1}_{CN,D}|$ goes beyond a pre-determined threshold.

Mismatch Detection. Our compatibility measure also returns 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 Fig. 4 present several mismatches, *e.g.*, s0 is initial while c1 is not, and their outgoing transition labels have the equal 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 $\textit{obs-comp}^k_{CN,D}$, where $D \in \{\rightarrow, \leftrightarrow\}$.

4.3 Analysis of Compatibility Measures

In this section, we first present how total protocol 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 implies that protocols are compatible if and only if their initial states are also compatible, *i.e.*, $CM_{CN,D}[I_1, I_2] = 1$. Such information is useful, *e.g.*, for automatically selecting available services in order to compose them.

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

```
global-res := 0, count := 0, matched-states := 0
for all s1 ∈ S1 do
    match := False
    for all s2 ∈ S2 do
        if CM_{CN,D}[s1, s2] ≥ t then
            global-res := global-res + CM_{CN,D}[s1, s2]
            match := True; count := count + 1
    if match = True then
        matched-states := matched-states + 1
if count ≠ 0 then
    global-res := (global-res / count) · (matched-states / |S1|)
return global-res
```

Algorithm 1: *global-comp*$(S_1, S_2, CM_{CN,D}, t)$

with high global compatibility degree enables to simplify further processing to resolve their interface incompatibility, *e.g.*, using service adaptation [13].

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. An alternative, shown in Algorithm 1, is to compute the global compatibility degree as the average of all compatibility degrees that are higher than or equal to a threshold t. To account for unmatched states, we multiply this average by the rate of states which have at least one possible matching with compatibility degree higher than t.

Algorithm 1 computes the global compatibility measure from one STS's point of view, and this works for the unidirectional compatibility notions. For the bidirectional compatibility notions, the global compatibility is computed as the average of the values returned by both functions *global-comp*$(S_1, S_2, CM_{CN,D}, t)$ and *global-comp*$(S_2, S_1, CM_{CN,D}, t)$.

Example 7. This example illustrates the computation of the global compatibility degree for the online store system of Fig. 4. Given a threshold $t = 0.7$ and the matrix of Table 4, the application of Algorithm 1 returns a global compatibility degree of 0.6. This rather low measure is justified by the state mismatch at c1 of the Customer protocol, which does not match with any state of the O-Store protocol (all compatibility values are below the threshold).

We can now also finish our example from the introduction, *cf.* Fig. 1. As this is a client-server setting, we compute compatibility using the *UR* notion. Using our tool Comparator and a threshold of 0.7 as above, we can compute the three MedServers' compatibility degrees with the Client service to 0.55, 0.85, and 0.76, respectively, hence preferring MedServer1 over the other two.

5 Tool Support

Our approach for measuring the compatibility degree of service protocols has been fully implemented in a prototype tool called Comparator [6]. The framework architecture is given in Fig. 5. The tool, implemented in Python, accepts as input two XML files corresponding to the service interfaces and an initial configuration,

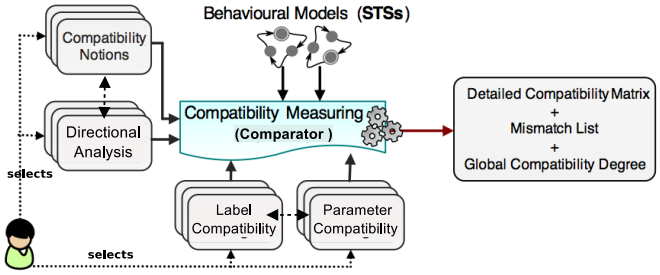


Fig. 5. Comparator architecture.

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 is highly modular, which makes easy its extension with new compatibility notions or other strategies for comparing message names and parameters. The tool can be used through a Web application [6].

We have validated our prototype tool on more than 110 examples, ranging from small ones, to experiment boundary cases, to real-world examples, *e.g.*, a car rental [7], a travel booking system [13], a video-on-demand application [22], music player system [22], a medical management system [4], and a multi-function device service [23]. Many case studies are available online at [6] and present the results of our approach for quantifying the compatibility.

Our experiments have shown that, even though efficiency was not our main concern in the application, Comparator can compute the compatibility degree of quite large systems in a reasonable time. Computation time depends on the size of the examples and the number of τ-transitions and loops.

To evaluate the preciseness of our compatibility measure, we have used the well-known precision and recall metrics [24] to estimate how much the measure automatically computed meets the expected result (see [6] for more details). For *compatible* protocols, our method yields a precision and recall of 100%. For incompatible protocols, we computed precision and recall metrics of 85% / 95% for the car rental example [7] and 91% / 100% for a flight advice system [6]; all other examples of our database returned values above 90% for both metrics.

6 Related work

We present several related approaches to measuring similarity or compatibility of interfaces. These are applied for service substitution and composition, respectively.
Protocol Traversal. The work in [25] measures the similarity of two computer viruses described using labelled transition systems (LTSs). It uses quantitative functions which are computed by a simple (not iterative) forward traversal of two LTSs. This work does not return the differences which distinguish one service from another, and there is no computation of a global similarity degree. In [27], the authors check the compatibility of two services described using the π-calculus. Here, two services are compatible if there is always at least one transition sequence

Table 1. A Summary of Approaches Based on Quantitative Behavioural Analysis.

		[15]	[25]	[11]	[1]	[27]	[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) 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		√	√	√	√	√	√	√

between them, until reaching final states. This notion is too weak since it does not guarantee deadlock-freeness for service composition. The authors compute the compatibility degree of two services as the average of the number of successful transition sequences. Neither detailed compatibility of different protocol states nor the mismatch list is returned.

Edit Distance. In [1, 11], the authors calculate the edit distance between a given *defective* service and *synthesised correct* services. They also detect the differences between two versions of one service interface described using finite state machines. The quantitative simulation measures the state similarity based on the analysis of outgoing transition labels without any semantic comparison of these label names, and there is no propagation of compatibility between neighbouring states.

Similarity Flooding. In [14], 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. The authors propose a set of metrics to measure correspondences between elements of data structures such as data schemas or data instances, described with LTSs. This work aims at assisting developers in matching elements of a schema by suggesting candidates. However, their tool does not enable fully automated matching. 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.

Quantitative Model Checking. The quantitative approach to service compatibility which we advocate here is related to recent quantitative approaches to model checking and verification. Here, Boolean notions of verification are replaced by distances, just as we do here for service compatibility. A general framework for such distance-based quantitative verification has been developed in [9].

We summarise in Table 1 the comparison of our approach with the closest related work.[7]

[7] BIS, SIM, WK, and OP abbreviate bisimulation, simulation, weak, and one path, respectively.

7 Conclusion

To the best of our knowledge, we suggest here the first generic framework for automatically quantifying the compatibility degree of service interfaces. Our measuring method relies on a compatibility flooding algorithm and is parametrised by different compatibility notions. In addition to computing the matrix and the global measure of compatibility, a list of mismatches is returned. Our computation always converges to a unique compatibility matrix.

Our proposal is fully supported by the Comparator tool which has been validated on many examples. We present some of these in a separate appendix; others are available online at [6]. The quality of results was measured using precision and recall, showing a low rate of false-positive results.

Our work has straightforward applications to service-related issues, *e.g.*, automatic discovery, selection, ranking, and composition. On a wider scale, our solution can be used to verify all software systems which can be described using STSs. In particular, Comparator has been used in a real-world case study in the context of the ITACA project [3] for service composition and adaptation. Comparator has also been integrated into a prototype tool, called Updator [19], which we implemented to deal with service evolution issues.

Our main perspective is to to apply our compatibility measuring approach for the automatic generation of adaptor protocols. This is a difficult issue, and only few attempts have been made in that direction. As an example, the techniques presented in [12] automate the generation of adaptation contracts for two services by combining graph search algorithms and heuristics. Unfortunately, this approach is costly and imprecise when services do not present only simple mismatches. We believe that by exploiting our method, we should obtain much better results in terms of computation time and relevance of adaptation contracts.

References

1. A. Aït-Bachir. Measuring similarity of service interfaces. In *ICSOC PhD Symposium*, vol. 421 of *CEUR Workshop Proceedings*, 2008.
2. S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. In *Architecting Systems with Trustworthy Components*, vol. 3938 of *LNCS*. Springer, 2006.
3. J. Cámara, J. A. 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 *ICSE'09*. IEEE, 2009.
4. J. Cámara, G. Salaün, C. Canal, and M. Ouederni. Interactive specification and verification of behavioural adaptation contracts. In *QSIC'09*. IEEE, 2009.
5. R. Cleaveland and O. Sokołsky. Equivalence and preorder checking for finite-state systems. *Handbook of Process Algebra*, 2001.
6. Comparator web page. `http://ouederni.perso.enseeiht.fr/tools.html`.
7. 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 *FACS*, vol. 215 of *ENTCS*, 2008.
8. F. Durán, M. Ouederni, and G. Salaün. A generic framework for n-protocol compatibility checking. *Sci. Comput. Program.*, 77(7-8):870–886, 2012.
9. U. Fahrenberg and A. Legay. The quantitative linear-time-branching-time spectrum. *Theor. Comput. Sci.*, 538:54–69, 2014.

10. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.
11. N. Lohmann. Correcting deadlocking service choreographies using a simulation-based graph edit distance. In *BPM'08*, vol. 5240 of *LNCS*. Springer, 2008.
12. J. A. Martín and E. Pimentel. Automatic generation of adaptation contracts. In *FOCLASA'08*, vol. 229(2) of *ENTCS*. Elsevier, 2009.
13. R. Mateescu, P. Poizat, and G. Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Trans. Software Eng.*, 38(4):755–777, 2012.
14. S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *ICDE*. IEEE, 2002.
15. S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE'07*. ACM Press, 2007.
16. H. R. M. Nezhad, G. Y. Xu, and B. Benatallah. Protocol-aware Matching of Web Service Interfaces for Adapter Development. In *WWW'10*. ACM, 2010.
17. M. Ouederni and G. Salaün. Tau be or not tau be? - a perspective on service compatibility and substitutability. In *WCSI'10*, vol. 37 of *EPTCS*, 2010.
18. M. Ouederni, G. Salaün, and E. Pimentel. Quantifying service compatibility: A step beyond the boolean approaches. In *ICSOC'10*, vol. 6470 of *LNCS*, 2010.
19. M. Ouederni, G. Salaün, and E. Pimentel. Client update: A solution for service evolution. In *SCC'12*. IEEE, 2011.
20. M. Ouederni, G. Salaün, and E. Pimentel. Measuring the compatibility of service interaction protocols. In *SAC'11*, vol. 2. ACM, 2011.
21. T. Pedersen, S. Patwardhan, and J. Michelizzi. Wordnet::similarity - measuring the relatedness of concepts. In *AAAI'04*. AAAI, 2004.
22. P. Poizat, G. Salaün, and M. Tivoli. An Adaptation-based Approach to Incrementally Build Component Systems. *ENTCS*, 182:155–170, 2007.
23. G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *SEFM'08*. IEEE, 2008.
24. G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
25. O. Sokołsky, S. Kannan, and I. Lee. Simulation-Based Graph Similarity. In *TACAS'06*, vol. 3920 of *LNCS*. Springer, 2006.
26. R. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.
27. Z. Wu, S. Deng, Y. Li, and J. Wu. Computing Compatibility in Dynamic Service Composition. *Knowledge Inf. Syst.*, 19(1):107–129, 2009.
28. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.
29. A. Zisman, G. Spanoudakis, and J. Dooley. A Framework for Dynamic Service Discovery. In *ASE'08*. IEEE, 2008.

Appendix: Applications

Our compatibility measure can be applied to several domains and can be used for verifying all software systems that can be described using STSs, *e.g.*, Web service and software components. For illustration purposes, we present the application of our detailed measure for solving some applications in service-oriented computing, namely service discovery [29], service adaptation [4] and service evolution [19].

A Service Discovery

Service discovery consists in selecting one or more services from a large number of candidates in order to integrate themselves in a second step into a service composition. Discovering compatible services is a hard and error-prone process if this is not assisted using automated techniques. In most cases, there is no compatible service and we are interested in finding the most compatible one in order to simplify, for instance, the adaptation techniques (see Section B) that can be used for solving mismatch issues.

Service discovery consists of the following steps: First, the providers advertise their services to a registry (*e.g.*, UDDI for Web services) where these services are stored; second, the requester asks the registry about the available services; third, the registry applies some search techniques in order to determine which service better matches the request.

In this setting, our global compatibility measure simplifies the discovery process in order to find services that are either perfectly compatible, *i.e.*, the global compatibility is equal to 1, or services that are mostly compatible, *e.g.*, having a high compatibility degree. In the latter case, the selection process relies on the global measure provided by our approach. The detailed compatibility measure, *e.g.*, the compatibility matrix and the mismatch list, can be helpful in a second step for resolving the mismatches (see, for instance, Sections B and C).

Example 8. Let us focus on the example given in Fig. 6 to illustrate the use of our global compatibility degree for service discovery. This example describes an on-line medical management system which handles patient appointments within a health care institution, either with general practitioners, or specialist doctors. The Client can first log on to a server by sending his/her user name and password (login!). Then, he/she asks for an appointment with a general practitioner (reqDoc!) and receives an appointment identifier. We present three services for the medical server, namely, MedServer0, MedServer1, and MedServer2 which are slightly different, yet they all can receive the patient name and password. Next, they can receive, and reply to, a request for an appointment with either a general practitioner (reqDoc?) or a specialist doctor (reqSpec?).

Considering the *UR* notion, the global compatibility degree for Client and the three possible medical servers MedServer0, MedServer1, and MedServer2 is equal to 0.55, 0.85, and 0.76, respectively. Note that without our measure, it is not obvious even for such a simple example to distinguish which service better matches the client requirements. However, using our results, we can decide to select MedServer1 instead of MedServer0 or MedServer2. Client presents only a parameter mismatch

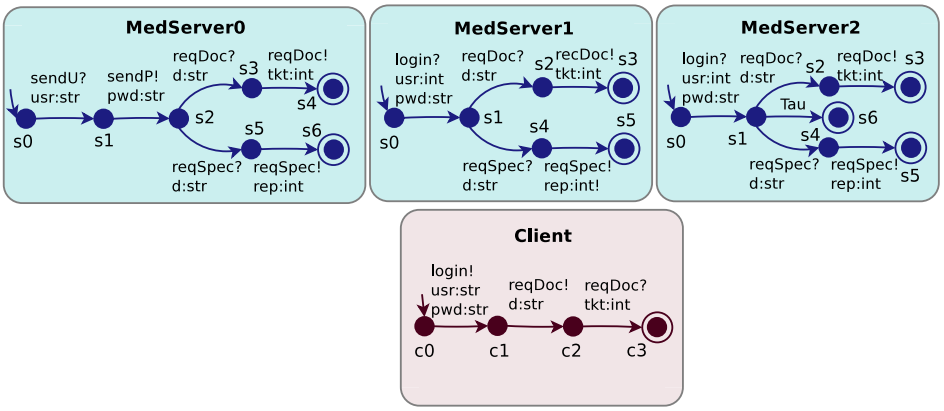


Fig. 6. STSs of a Medical Management System.

with MedServer1 (the usr parameter is defined using a different type in the two interfaces). In contrast, Client can deadlock with MedServer2, *i.e.*, due to the internal τ action (corresponding to a termination after a timeout). Client and MedServer0 present several mismatches (message names and number). Generally speaking, the highest global compatibility degree corresponds to the lowest number of mismatches between the two interfaces. In our example, the parameter issue in the client protocol is much easier to resolve than the deadlock problem, particularly because we cannot modify the servers, assuming a black-box hypothesis.

B Service Adaptation

Adaptation aims at computing an intermediate service – called adaptor – to resolve mismatches presented between services interacting with each other. An adaptor describes composition constraints and adaptation requirements among these services and is built from abstract descriptions – called contracts – of how interface mismatches can be worked out.

The work given in [4] proposes a graphical environment – called ACIDE – for interactive contract specification using our compatibility measure. Note that, so far, ACIDE only considers the *UR* compatibility notion. The graphical notation for a service interface in ACIDE includes a representation of behavioural models (STSs) and a collection of ports. Each label on the STS corresponds to a port in the graphical description of the interface. Ports include a data port for each parameter contained in the parameter list of the label. Correspondences between the different service interfaces are represented as port bindings and data port bindings. Starting from the graphical representation of the interfaces, the architect can build a contract between them by successively connecting ports and data ports. This results in the creation of bindings which specify how the interactions should be carried out between the services.

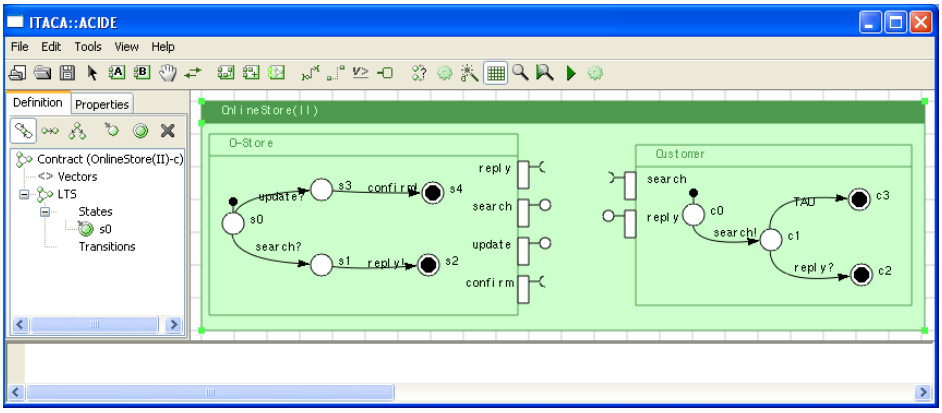


Fig. 7. Online Store (II) Graphical Presentation in ACIDE.

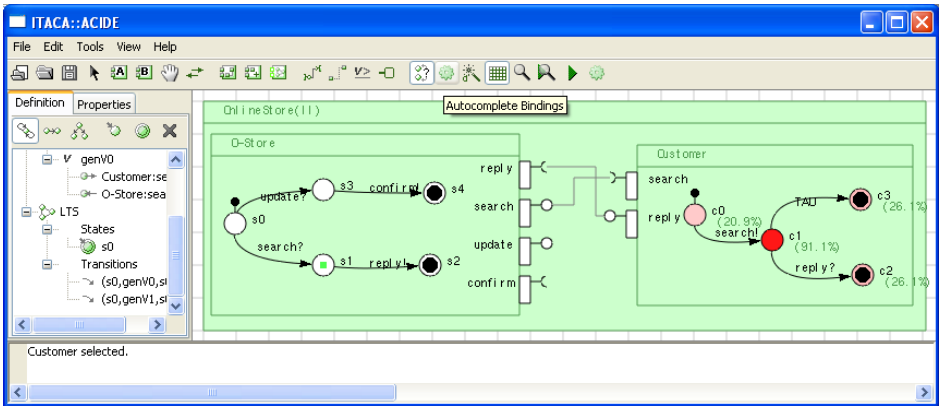


Fig. 8. Label-based Matching in ACIDE.

Example 9. Fig. 7 shows the graphical presentation of the STSs given in Fig.6 in the main paper.[8]

In order to specify the adaptation contract in ACIDE, our compatibility measure can be used in different ways. Firstly, it is possible to automatically generate port bindings for labels that perfectly match.

Example 10. Considering the STSs presented in Fig.6 in the main paper, the result of automatic port binding is shown in Fig. 8.

In ACIDE, the user can also select a transition label l in one protocol (we call s its source state in the rest of this paragraph), and then Comparator is used to

[8] In this example, the Wordnet similarity package is omitted and the seek! label is renamed into search!.

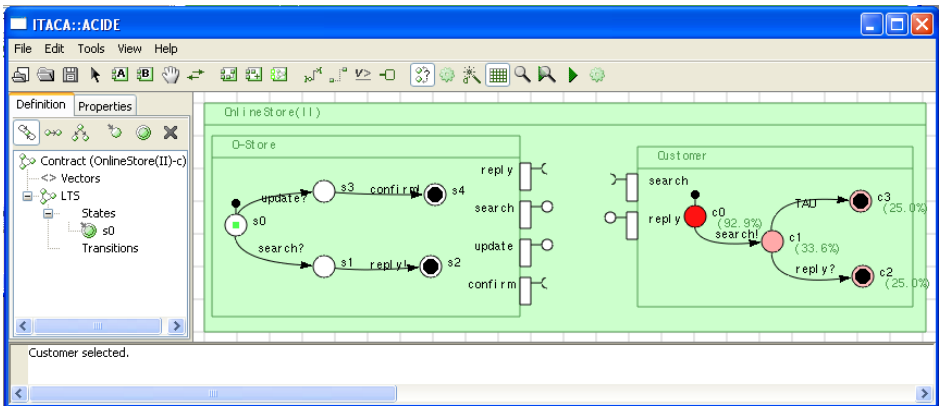


Fig. 9. State-based Matching in ACIDE.

return the best label matching in the other protocol. Note that Comparator can return several possible matchings for either states or labels that are compatible with each other in a partner service. This is similar to the simulation (preorder) relation in concurrency theory [5] where several states can be simulated by one state. In this case, designer intervention is required to decide which bindings should be kept (See Example 12 for illustration). To do so, two functions have been implemented where: (i) the first function labels all states in the other protocol with compatibility measures between s and every state in the partner interface, and (ii) the second function seeks the highest value (s, s_j) in the matrix (where s_j is a partner state) and returns the label going out from s_j the most compatible with l. These functions can be completed with other alternatives such as returning the best label matching for each state in the partner, or for each state whose compatibility measure with respect to s is higher than a threshold t. To highlight these results in the graphical interface, ACIDE does not only display the compatibility measures but also colour in red the best matchings.

Example 11. Fig. 9 shows the binding result returned for state s0 in the O-Store protocol with all states of the Customer protocol given in Fig.6 in the main paper. As we can observe, s0 matches the initial state c0 on Customer with which it has the highest compatibility value equal to 92.9%. Based on this highest state compatibility measure, the labels going out from s0 will be matched with the most compatible labels going out from c0. We recall that label binding can be done manually based on the measure of label compatibility or following the automatic binding as presented in Fig. 8.

Example 12. Let us focus on the example given in Fig. 10 to illustrate how our global compatibility degree can yield a *false-positive* evaluation considering service adaptation. This example describes an on-line medical management system [4] which handles patient appointments within a health care institution, either with general practitioners or specialist doctors. We present two versions of the Medical

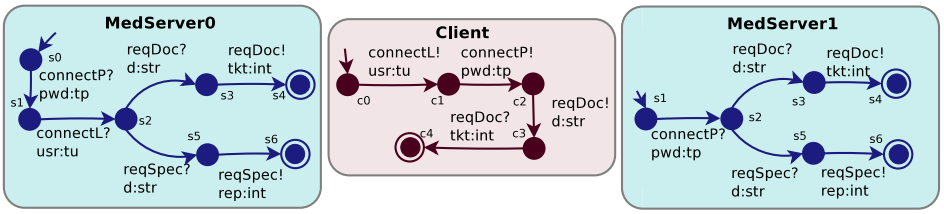


Fig. 10. STSs of a Medical Management System.

Server, namely MedServer0 and MedServer1 which are slightly different. MedServer0 starts with receiving the patient login (connectL?) and then his/her password (connectP?), while MedServer1 expects to receive the password at its initial state. Next, both can receive, and reply to, a request for an appointment with either a general practitioner (reqDoc?) or a specialist doctor (reqSpec?). The Client sends his/her password followed by his/her user name. Then, the Client asks for an appointment with a general practitioner (reqDoc!) and receives an appointment identifier.

Considering the *UR* notion, the global compatibility degree for MedServer0 and MedServer1 with Client is equal to 0.49 and 0.68, respectively. Based on our compatibility measure, one would prefer to adapt MedServer1 with Client. However, the adaptation does not work in this case to resolve the missing parameter issue (*i.e.*, usr is missing in MedServer1 but present in Client). In the other case, although both MedServer0 and Client return a lower value, these can be adapted using message reordering techniques such as those proposed in [13].

C Service Evolution

We now present another application of our compatibility measure, namely, service evolution. Although many efforts have been devoted by researchers for automating service adaptation, this solution cannot be applied in some cases, *e.g.*, the types of sent and received parameters mismatch, yet type conversion is not allowed. Therefore, services interfaces should be changed in order to work out such interoperability issues.

The remainder of this section shows how our compatibility measure is used to resolve the interface mismatches and make a service and its client (user application) compatible. To this end, we consider the *UC* notion for illustration purposes such that the client and the service represent the complemented and the complementer partners, respectively. In the following, we present the intuition behind the application and we refer the readers to [19] for technical details. The application is automated by a prototype tool (called Updator).

Overview of the Client Update Process. In order to resolve the incompatibility issues, an automated process to change the client interface has been proposed, and this is shown in Fig. 11. In step 1, it computes the compatibility measure which compares both interfaces. Then, step 2 relies on the resulting compatibility matrix to generate an interface mapping tree which describes the best state matching on both interfaces. Based on the analysis of the mapping tree, the client interface is

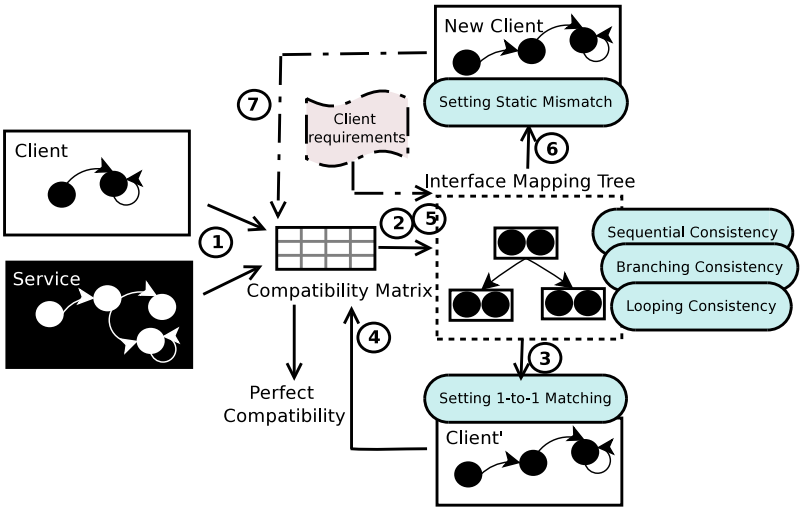


Fig. 11. Overview of our Client Update Process

modified as follows. In step 3, behavioural mismatches are resolved, *e.g.*, addition of missing transitions or removal of mismatching transitions to and from the client interface. Steps 4 and 5 compute the interface compatibility and the mapping tree again in order to take the changes made in the previous step into account. The resulting mapping tree describes 1-to-1 state mapping, where every state on one interface has its corresponding matching state on the other interface. By doing so, in step 6, the updated tree is considered to resolve the static mismatches, which can be presented between labels of transitions going out from matched states. Lastly, after resolution of interface mismatches, step 7 computes the compatibility measure to validate that the updated client has become compatible with its service interface. At this very last step, a new iteration starts from step 2 if the interface compatibility is achieved yet the user requirements are not satisfied. Otherwise, the update process terminates here.

Note that the interface mapping tree describes a set of linked nodes where each node represents the best matching of a client state with one state among those on the service interface. Furthermore, each node is linked to its parent and children nodes.

The update process can be parametrised by a set of user requirements to prevent undesirable behaviours that the designer does not want to appear in the new client interface. These requirements consist of a set of messages which must not appear in the updated interface.

Example 13. Let us illustrate the computation of an interface mapping tree from a compatibility matrix and some user requirements. We give in Fig. 12 a simplified example of a database management system where a User (complemented) can access an online Database (complementer) to search data or make an update, and waits for its acknowledgement. The database service first receives a request for an

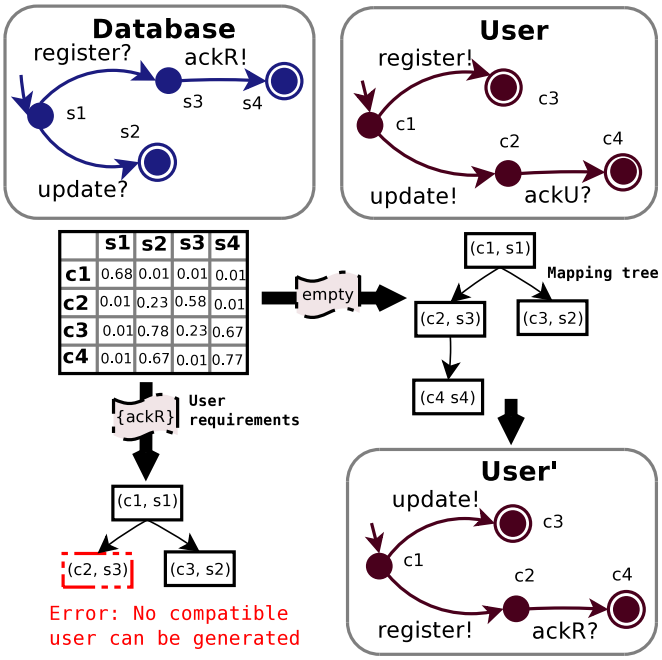


Fig. 12. Database Management System.

update or a registration to be acknowledged. Unfortunately, the protocols are not compatible.

Let us first suppose an empty set of user requirements. Each node in the mapping tree represents a User state with its best state match among those in Database. the update process changes the User protocol into User', which is compatible with Database protocol.

We now suppose a non-empty set of user requirements, which is equal to {ackR}, *i.e.*, the user does not want to receive the registration acknowledgement. The computation of the mapping tree returns a deadlock node (represented by the dashed rectangle in Fig. 12). This node does not have any child because the message ackR! going out from state s3 cannot be matched with any message at state c2 due to the restriction made by the user requirements. Thus, no compatible user interface can be generated using this interface mapping tree.

Resolution of Behavioural Mismatches. An interface mapping tree is used for resolving the behavioural mismatches. The techniques aim at ensuring the 1-to-1 state matching from the client viewpoint. To do so, several systematic changes can be made using pre-defined patterns. The very first pattern is referred to as *add/remove states* and enables us to check whether one or more client states must be removed if they have no match on the service interface. This pattern makes it also possible to add states to the client interface if there exist service states with no match. The top of Fig. 13 illustrates an example where the service's state s2 does not have any match in Int_{clt}. Note that states s1 and s3 match states c1 and

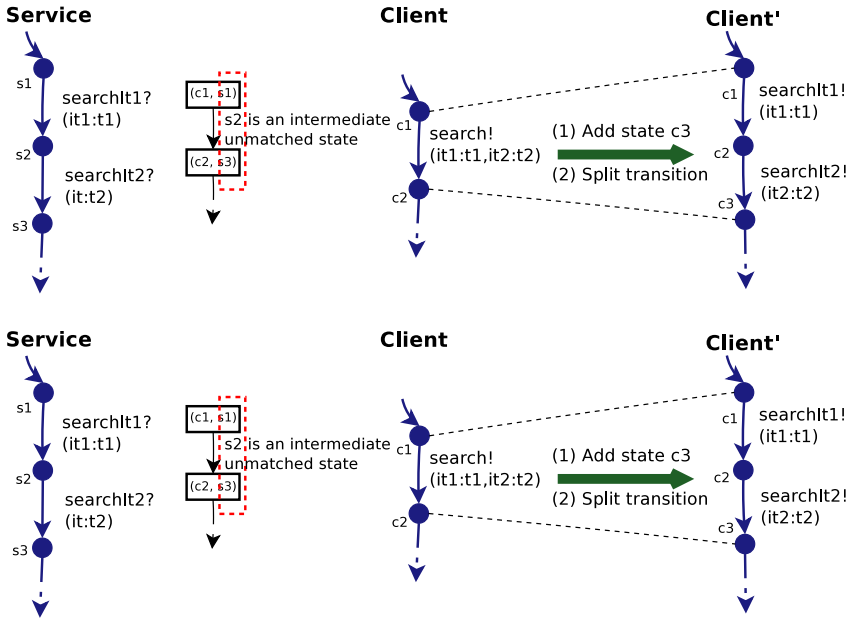


Fig. 13. Patterns: Add State and Split Transitions (top), Add State and Split Transitions (bottom)

c2, respectively. Thus, since s2 is the successor of s1 and also the predecessor of s3, the client change consists in adding a new state to be matched with s2.

Add/remove states may lead to another pattern called *merge/split transitions* where transitions can be removed or added. This behavioural change deals with the protocol information in order to keep it coherent wrt. the predecessor and successor behaviour of a removed or added transition. For instance, the bottom of Fig. 13 shows that there are two search transitions on the service protocol matching one search transition on the client interface. To resolve this mismatch, the client search transition is split into two transitions. Regarding the rest of the client protocol, here the client behaviour starting from state c2, the link of such a state with its successors is updated, considering the added state c3.

Resolution of Static Mismatches. The mismatch list computed by Comparator is used to resolve the static mismatches. Such mismatches concern either state nature or labels. Here, three possible changes are applied: (i) unifying the nature of matched states; (ii) renaming transition labels; or (iii) updating the signature and the alphabets with new labels and operation profiles.

Appendix: Proof of Convergence

In this appendix we provide a detailed proof that our algorithm for computing behavioural compatibility between services converges in a finite number of steps, regardless of its input.

We show the proof for unidirectional complementarity only; it is similar for the other notions. For ease of reference, we repeat the relevant equations from Section 4.2:

$$CM^k_{UC,\to}(s_1,s_2) = \frac{CM^{k-1}_{UC,\to}(s_1,s_2) + \textit{state-comp}^k_{UC,\to}(s_1,s_2)}{2} \tag{1}$$

$$\textit{state-comp}^k_{UC,\to}(s_1,s_2) = \frac{w_1 \cdot \textit{fw-propag}^k_{UC,\to}(s_1,s_2) + w_2 \cdot \textit{bw-propag}^k_{UC,\to}(s_1,s_2) + nat(s_1,s_2)}{w_1+w_2+1} \tag{2}$$

$$\textit{fw-propag}^k_{UC,\to}(s_1,s_2) = \begin{cases} \dfrac{\sum_{(s_1,\tau,s_1')} \textit{fw-propag}^k_{UC,\to}(s_1',s_2) + \textit{obs-comp}^k_{UC,\to}(s_1,s_2)}{|tau(s_1)|+1} \\ \qquad \text{if } tau(s_1) \neq \varnothing \\ \dfrac{\sum_{(s_2,\tau,s_2')} \textit{fw-propag}^k_{UC,\to}(s_1,s_2') + \textit{obs-comp}^k_{UC,\to}(s_1,s_2)}{|tau(s_2)|+1} \\ \qquad \text{otherwise} \end{cases} \tag{3}$$

$$\textit{obs-comp}^k_{UC,\to}(s_1,s_2) = \begin{cases} 1 & \text{if } \textit{sum}^k_{UC,\to}(s_2,s_1,Fw(s_2),Fw(s_1)) = |Fw(s_2)| \\ & \text{and } (s_2,s_1) \in DF \\ 0 & \text{if } (s_2,s_1) \notin DF \\ \dfrac{\textit{sum}^k_{UC,\to}(s_2,s_1,Fw(s_2),Fw(s_1))}{\max(|Fw(s_2)|,|Fw(s_1)|)} & \text{otherwise} \end{cases} \tag{4}$$

$$\textit{sum}^k_{UC,\to}(s_1,s_2,T_1,T_2) = \begin{cases} \sum\limits_{(s_1,l_1,s_1')\in T_1} \max\limits_{(s_2,l_2,s_2')\in T_2} (\textit{lab-comp}(l_1,l_2)\, CM^{k-1}_{UC,\to}(s_1',s_2')) \\ \qquad \text{if } |Fw(s_1,T_1)| \neq 0 \text{ and } |(Fw(s_2,T_2)| \neq 0 \\ 0 \qquad \text{otherwise} \end{cases} \tag{5}$$

Let $F : [0,1]^{n\times n} \to [0,1]^{n\times n}$ be the function defined by equations (1) to (5) above, i.e.

$$F(M)(s_1,s_2) = \frac{M(s_1,s_2) + \textit{state-comp}_{UC,\to}(M)(s_1,s_2)}{2} \tag{6}$$

$$\textit{state-comp}_{UC,\to}(M)(s_1,s_2) = \frac{w_1\, \textit{fw-propag}_{UC,\to}(M)(s_1,s_2) + w_2\, \textit{bw-propag}_{UC,\to}(M)(s_1,s_2) + nat(s_1,s_2)}{w_1+w_2+1} \tag{7}$$

$$\textit{fw-propag}_{UC,\to}(M)(s_1,s_2) = \begin{cases} \dfrac{\sum_{(s_1,\tau,s_1')} \textit{fw-propag}_{UC,\to}(M)(s_1',s_2) + \textit{obs-comp}_{UC,\to}(M)(s_1,s_2)}{|tau(s_1)|+1} \\ \qquad \text{if } tau(s_1) \neq \varnothing \\ \dfrac{\sum_{(s_2,\tau,s_2')} \textit{fw-propag}_{UC,\to}(M)(s_1,s_2') + \textit{obs-comp}_{UC,\to}(M)(s_1,s_2)}{|tau(s_2)|+1} \\ \qquad \text{otherwise} \end{cases} \tag{8}$$

$$\textit{obs-comp}_{UC,\to}(M)(s_1,s_2) = \begin{cases} 1 & \text{if } \textit{sum}_{UC,\to}(M)(s_2,s_1,Fw(s_2),Fw(s_1)) = |Fw(s_2)| \\ & \quad \text{and } (s_2,s_1) \in DF \\ 0 & \text{if } (s_2,s_1) \notin DF \\ \frac{\textit{sum}_{UC,\to}(M)(s_2,s_1,Fw(s_2),Fw(s_1))}{\max(|Fw(s_2)|,|Fw(s_1)|)} & \text{otherwise} \end{cases} \tag{9}$$

$$\textit{sum}_{UC,\to}(M)(s_1,s_2,T_1,T_2) = \begin{cases} \sum\limits_{(s_1,l_1,s_1')\in T_1} \max\limits_{(s_2,l_2,s_2')\in T_2} (\textit{lab-comp}(l_1,l_2)\, M(s_1',s_2')) \\ \quad \text{if } |Fw(s_1,T_1)| \neq 0 \text{ and } |(Fw(s_2,T_2)| \neq 0 \\ 0 \quad \text{otherwise} \end{cases} \tag{10}$$

Let $\lambda = \frac{1}{2}(1+\frac{w_1+w_2}{w_1+w_2+1})$, then $\lambda < 1$. We show that F is λ-Lipschitz continuous, hence a contraction. Banach's fixed-point theorem then ensures that F has a unique fixed point and that the iteration given by equations (6) to (10) in finitely many steps reaches the fixed point with arbitrary precision.

We use the max-metric for matrices, i.e. $\|M\| = \max_{i,j} |M^{ij}|$. Let $M_1, M_2 \in [0,1]^{n\times n}$, then

$$\begin{aligned} &\|F(M_1) - F(M_2)\| \\ &\quad \leq \tfrac{1}{2} \max_{i,j} |M_1^{ij} - M_2^{ij}| + \tfrac{1}{2} \|\textit{state-comp}_{UC,\to}(M_1) - \textit{state-comp}_{UC,\to}(M_2)\| \end{aligned}$$

Now

$$\begin{aligned} &(\textit{state-comp}_{UC,\to}(M_1) - \textit{state-comp}_{UC,\to}(M_2))^{ij} = \\ &\qquad \tfrac{1}{w_1+w_2+1}\Big(w_1(\textit{fw-propag}_{UC,\to}(M_1)^{ij} - \textit{fw-propag}_{UC,\to}(M_2)^{ij}) \\ &\qquad\qquad + w_2(\textit{bw-propag}_{UC,\to}(M_1)^{ij} - \textit{bw-propag}_{UC,\to}(M_2)^{ij})\Big), \end{aligned}$$

and as the formulas for $\textit{fw-propag}_{UC,\to}$ and $\textit{bw-propag}_{UC,\to}$ are entirely analogous, we can assume that

$$\begin{aligned} &|(\textit{state-comp}_{UC,\to}(M_1) - \textit{state-comp}_{UC,\to}(M_2))^{ij}| \leq \\ &\qquad \tfrac{w_1+w_2}{w_1+w_2+1} |\textit{fw-propag}_{UC,\to}(M_1)^{ij} - \textit{fw-propag}_{UC,\to}(M_2)^{ij}|, \end{aligned}$$

hence,

$$\begin{aligned} \|F(M_1) - F(M_2)\| &\leq \tfrac{1}{2} \max_{i,j} |M_1^{ij} - M_2^{ij}| \\ &\quad + \tfrac{1}{2} \tfrac{w_1+w_2}{w_1+w_2+1} \|\textit{fw-propag}_{UC,\to}(M_1) - \textit{fw-propag}_{UC,\to}(M_2)\|. \end{aligned}$$

To obtain an upper bound for $\|\textit{fw-propag}_{UC,\to}(M_1) - \textit{fw-propag}_{UC,\to}(M_2)\|$, we note that $\textit{fw-propag}_{UC,\to}(M)(s_1,s_2)$ essentially computes a weighted average of $\textit{obs-comp}_{UC,\to}(M)(s_1',s_2')$ for all states s_1', s_2' reachable from s_1, resp. s_2, by

sequences of τ-transitions: assuming $tau(s_2) = \emptyset$ for now, we have

$$\begin{aligned}
&\textit{fw-propag}_{UC,\to}(M)(s_1, s_2) \\
&\quad = \tfrac{\textit{obs-comp}_{UC,\to}(M)(s_1,s_2)}{|tau(s_1)|+1} + \sum_{s_1 \xrightarrow{\tau} s_1'} \tfrac{\textit{fw-propag}_{UC,\to}(M)(s_1',s_2)}{|tau(s_1)|+1} \\
&\quad = \tfrac{\textit{obs-comp}_{UC,\to}(M)(s_1,s_2)}{|tau(s_1)|+1} + \sum_{s_1 \xrightarrow{\tau} s_1'} \tfrac{\textit{obs-comp}_{UC,\to}(M)(s_1',s_2)}{(|tau(s_1)|+1)(|tau(s_1')|+1)} \\
&\qquad + \sum_{s_1 \xrightarrow{\tau} s_1' \xrightarrow{\tau} s_1''} \tfrac{\textit{fw-propag}_{UC,\to}(M)(s_1'',s_2)}{(|tau(s_1)|+1)(|tau(s_1')|+1)} \\
&\quad = \cdots ,
\end{aligned}$$

thus,

$$\begin{aligned}
&\textit{fw-propag}_{UC,\to}(M_1)(s_1, s_2) - \textit{fw-propag}_{UC,\to}(M_2)(s_1, s_2) \\
&\quad = \tfrac{\textit{obs-comp}_{UC,\to}(M_1)(s_1,s_2) - \textit{obs-comp}_{UC,\to}(M_2)(s_1,s_2)}{|tau(s_1)|+1} \\
&\qquad + \sum_{s_1 \xrightarrow{\tau} s_1'} \tfrac{\textit{obs-comp}_{UC,\to}(M_1)(s_1',s_2) - \textit{obs-comp}_{UC,\to}(M_2)(s_1',s_2)}{(|tau(s_1)|+1)(|tau(s_1')|+1)} + \cdots + \\
&\qquad + \sum_{s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_1^{(n)}} \tfrac{\textit{obs-comp}_{UC,\to}(M_1)(s_1^{(n)},s_2) - \textit{obs-comp}_{UC,\to}(M_2)(s_1^{(n)},s_2)}{(|tau(s_1)|+1)\cdots(|tau(s_1^{(n-1)})|+1)} .
\end{aligned}$$

Hence, also lifting the assumption that $tau(s_2) = \emptyset$, we see that

$$\begin{aligned}
&|\textit{fw-propag}_{UC,\to}(M_1)(s_1, s_2) - \textit{fw-propag}_{UC,\to}(M_2)(s_1, s_2)| \leq \\
&\qquad \max_{s_1', s_2'} |\textit{obs-comp}_{UC,\to}(M_1)(s_1', s_2') - \textit{obs-comp}_{UC,\to}(M_2)(s_1', s_2')| .
\end{aligned}$$

We have shown that

$$\begin{aligned}
&\|F(M_1) - F(M_2)\| \leq \tfrac{1}{2} \max_{i,j} |M_1^{ij} - M_2^{ij}| \\
&\qquad + \tfrac{1}{2} \tfrac{w_1+w_2}{w_1+w_2+1} \max_{i,j} |\textit{obs-comp}_{UC,\to}(M_1)^{ij} - \textit{obs-comp}_{UC,\to}(M_2)^{ij}| .
\end{aligned}$$

Now to bound $|\textit{obs-comp}_{UC,\to}(M_1)(s_1, s_2) - \textit{obs-comp}_{UC,\to}(M_2)(s_1, s_2)|$ from above, we see that its maximum is attained when both values fall in the last case

of (9), and then

$$
\begin{aligned}
&|\textit{obs-comp}_{UC,\to}(M_1)(s_1,s_2) - \textit{obs-comp}_{UC,\to}(M_2)(s_1,s_2)| \\
&= \tfrac{|sum_{UC,\to}(M)(s_2,s_1,Fw(s_2),Fw(s_1)) - sum_{UC,\to}(M)(s_2,s_1,Fw(s_2),Fw(s_1))|}{\max(|Fw(s_2)|,|Fw(s_1)|)} \\
&= \tfrac{1}{\max(|Fw(s_2)|,|Fw(s_1)|)} \sum_{s_2 \xrightarrow{l_2} s_2'} \Big| \max_{s_1 \xrightarrow{l_1} s_1'} \textit{lab-comp}(l_1,l_2)\, M_1(s_2',s_1) \\
&\qquad\qquad - \max_{s_1 \xrightarrow{l_1} s_1'} \textit{lab-comp}(l_1,l_2)\, M_2(s_2',s_1) \Big| \\
&\le \tfrac{1}{\max(|Fw(s_2)|,|Fw(s_1)|)} \sum_{s_2 \xrightarrow{l_2} s_2'} \max_{s_1 \xrightarrow{l_1} s_1'} \textit{lab-comp}(l_1,l_2) \big| M_1(s_2',s_1) - M_2(s_2',s_1) \big| \\
&\le \tfrac{1}{n} \sum_j \max_i |M_1^{ij} - M_2^{ij}| \le \max_{i,j} |M_1^{ij} - M_2^{ij}| \,.
\end{aligned}
$$

This now entails that

$$
\begin{aligned}
\|F(M_1) - F(M_2)\| &\le \tfrac{1}{2} \max_{i,j} |M_1^{ij} - M_2^{ij}| + \tfrac{1}{2} \tfrac{w_1+w_2}{w_1+w_2+1} \max_{i,j} |M_1^{ij} - M_2^{ij}| \\
&= \lambda \|M_1 - M_2\| \,,
\end{aligned}
$$

which is what was to be shown.