Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 255 (2009) 65–81
www.elsevier.com/locate/entcs

Checking Protocol Compatibility using Maude

Francisco Durán[1] Meriem Ouederni[2] Gwen Salaün[3]

Dept. of Computer Science
University of Málaga
Málaga, Spain

Abstract

Checking compatibility of services accessed through their behavioural interfaces is a crucial issue in Service Oriented Computing which aims at building new systems from existing software services. In this paper, we consider a model of services which takes value passing and non-observable actions into account. We propose an approach to check in a unified way several compatibility notions between two service protocols using the rewriting logic system Maude. In particular, we illustrate our approach with three widely used compatibility notions, namely opposite behaviours, unspecified receptions and deadlock freeness. These notions as well as several strategies to handle non-observable actions have been formalised and fully implemented into a prototype tool which is able to automatically detect whether two services are compatible, and return a counterexample if they are not.

Keywords: Symbolic Transition Systems, non-observable actions, compatibility, Maude.

1 Introduction

The main principle of Component Based Software Engineering (CBSE) and Service-Oriented Computing (SOC) is the reuse of existing software components or Web services [4] when building new systems in order to save time and efforts. In this context, interoperability issues must be avoided and reused services have to fit with other services in the system being developed. Therefore, compatibility checking is a crucial concern to ensure whether services can interoperate successfully. However, verifying compatibility is especially difficult when the model of services takes interaction protocols into account which is essential [20] to avoid erroneous behaviours or deadlock situations when executing together a set of services.

[1] Email: duran@lcc.uma.es
[2] Email: meriem@lcc.uma.es
[3] Email:salaun@lcc.uma.es

[4] In the remainder of this paper, we will refer to both software component and Web service using the generic term *service*.

doi:10.1016/j.entcs.2009.10.025

Numerous compatibility notions have already been proposed based upon several description models of service interfaces, see for instance [23,9,20,2] for automata, [4,10] for π-calculus, [14,16,17] for Petri nets and [1,5] for state machines. Most of these approaches focus on a unique compatibility notion defined with respect to a specific application area, see for instance [17] for service composition, [1,5,14] for service substitution, [10] for service choreography, etc. Compared to existing works, new contributions of our proposal are the following: (i) our model takes into account value-passing as well as non-observable or internal actions (τ actions), and considers different strategies to handle them; (ii) we propose a framework where several notions of compatibility can be checked automatically, and this framework can be extended with new compatibility notions and new strategies to handle internal actions. Note that internal actions in interface models are very important because some services can be compatible from an observable point of view, but their execution will behave erroneously due to uncontrolled internal behaviours.

Our model of service interfaces allows to specify interaction protocols (messages and their application order). Value-passing and internal actions are also considered. Our framework provides a uniform and systematic way of checking different notions of compatibility, considering different strategies to deal with internal actions. Here, we focus on three widely used compatibility notions, namely opposite behaviours [21], unspecified receptions [3,23], and deadlock freeness [2]. But other notions are already available in our framework and additional ones could be considered in the future. Moreover, we consider three different τ handling strategies, namely the strong, weak, and trace strategies introduced by Robin Milner in the context of equivalence checking in the 80s [21], and similarly, additional ones could be added to the framework as well. Note that it is important considering different strategies, since each of them makes sense depending on the objective of the compatibility check. Thus, for example, a trace strategy is well suited if one wants to compare external behaviours, whereas a weak strategy allows to detect some subtle errors when composing services. A strong strategy is required when two services must exactly match from an internal and external point of view as it is the case while substituting one service for another.

Our solution for checking protocol compatibility is independent of any specific application, and therefore can be useful for several issues, such as automatic service composition, software adaptation, service substitution, service discovery, code reengineering, etc. Our proposal has been implemented in the rewriting logic system Maude [7]. Its powerful rewriting engine and searching capabilities open a very wide range of possibilities. Moreover, its expressivity has allowed us to implement the different notions and strategies at a very high level of abstraction, without compromising efficiency, and allowing such a flexible and extensible general framework.

The remainder of this paper is structured as follows. Section 2 formalises our model of services. Section 3 introduces some preliminaries and the notions of compatibility we use in this paper for illustration purposes. In Section 4, we present how service compatibility is checked using Maude. Section 5 presents a comparison with related approaches. Finally, Section 6 draws up some conclusions.

$$\frac{(s \xrightarrow{\tau} s') \in T}{\langle s, E\rangle \xrightarrow{\tau}_b \langle s', E\rangle} \quad \text{(TAU)}$$

$$\frac{(s \xrightarrow{a!v} s') \in T \quad v' = ev(v, E)}{\langle s, E\rangle \xrightarrow{a!v'}_b \langle s', E\rangle} \quad \text{(EM)} \qquad \frac{(s \xrightarrow{a?x} s') \in T}{\langle s, E\rangle \xrightarrow{a?x}_b \langle s', E\rangle} \quad \text{(REC)}$$

Fig. 1. Operational Semantics of one STS

2 Model of Services

We assume that service interfaces are equipped both with a signature (set of required and provided operations) and a protocol represented by a *Symbolic Transition System* (STS). Communication between services is represented using *events* relative to the emission and reception of messages corresponding to operation calls. Events may come with a list of parameters whose types respect the operation signatures. In our model, a *label* is either the internal action τ or a tuple $(m, d, (p_1, \ldots, p_n))$ where m is the message name, d stands for the communication direction (either an emission ! or a reception ?), and $(p_1, \ldots, p_n)$ is either a list of data terms if the label corresponds to an emission, or a list of variables if the label is a reception.

Definition 2.1 [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, $F \subseteq S$ are the final states, and $T : S \times A \times S$ is the transition function.

In the remainder of this paper, 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 the label.

Our STS is a reduced version of STG (Symbolic Transition Graph) introduced in [15]. Here, guards are abstracted as τ transitions, which denote internal (non-observable) activity to the service. The operational semantics of one STS ($\rightarrow_b$) is defined with three rules (TAU, EM, REC, Figure 1).

The operational semantics of n ($n > 1$) STSs ($\rightarrow_c$) is formalised using a synchronous communication rule (COM, Figure 2) in which value-passing and variable substitutions rely on a late binding semantics [19], and an independent evolution rule (INE$_\tau$, Figure 2) . $\{as_1, \ldots, as_n\}$ is a set of couples $\langle s_i, E_i\rangle$, where each couple $\langle s, E\rangle$ is composed by an active state $s \in S$ and a data environment E. A data environment is a set of couples $\langle x, v\rangle$ where x is a variable and v a ground value. We use a function *type* which returns the type of a variable or a value, and we define the environment update "$\oslash$", and the evaluation function ev as follows:

$$E \oslash \langle x, v\rangle \triangleq E(x) = v$$

$$ev(E, x) \triangleq E(x)$$

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

The STS formal model has been chosen because it is simple, graphical, and it can be easily derived from existing implementation platforms' languages (see for instance [12,22,11] where such abstractions for Web services were used for verifi-

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

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

Fig. 2. Operational Semantics of n STSs

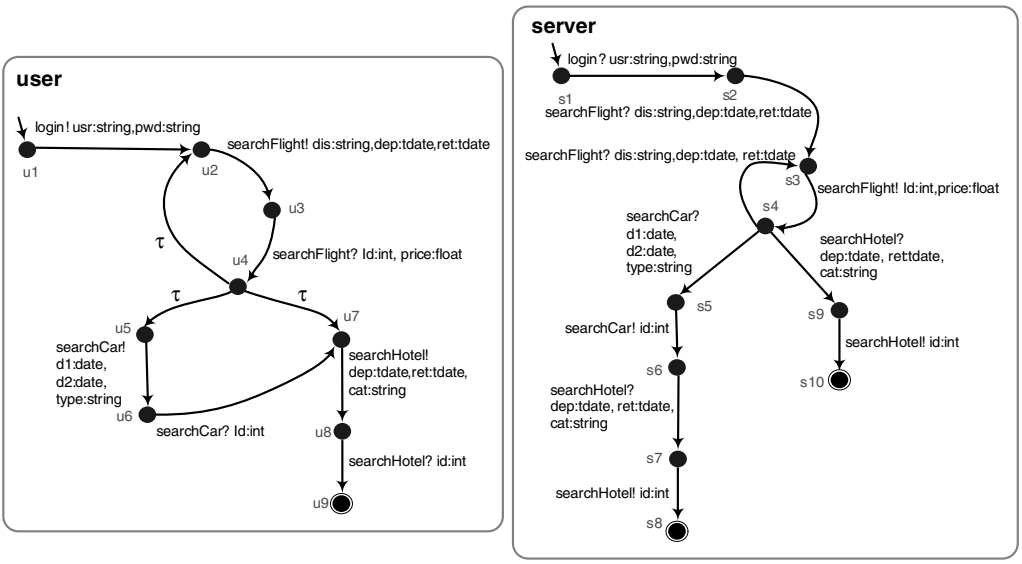

Fig. 3. User and Server Behavioural Interfaces

cation, composition or adaptation purposes). For space reasons, in the rest of the paper, we will describe service interfaces only with their STSs. Signatures will be left implicit, yet they can be inferred from the typing of arguments (made explicit here) in STS labels.

Example. We use as running example a travel advice system which gives information for travelers. As it can be observed in Figure 3, the user interface can first log on to a server by sending its user name and password (login!). Then, it can request (searchFlight!) and receive (searchFlight?) a flight identifier and price according to the wished destination and departure and return dates. Depending on its preferences (internal choice specified with τ transitions in the user protocol), the user can either start a new flight request, ask for a hotel advice (searchHotel!), and then it receives the answer (searchHotel?), or ask for a car advice (searchCar!), get the answer (searchCar?), and then start a hotel advice request. The server service starts by receiving a user name and password (login?). Next, this service can receive and reply several flight requests until a request for a car and/or a hotel advice is received and replied.

3 Service Protocol Compatibility

We start with some definitions useful for the formalisation of service protocol compatibility that we will present in Section 3.4. Since our model considers τ actions, the definition of different compatibility notions depends on how these non-observable actions are handled. In this paper, we consider three widely used τ handling strategies, namely strong, weak and trace [21], and the compatibility notions can be checked with respect to each of these strategies. Moreover, we formalise the different notions of protocol compatibility using the concepts of *set of global reachable states* and *transition compatibility*.

3.1 Preliminaries

In this paper, we focus on the case of two services described using STSs $STS_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, 2\}$. We define a function $out_T(s)$ which returns the set of transitions going out from the state s in the transition set T. Last, functions $reachable_T(s)$ and $reachable_{T,l}(s)$ return the set of states which can be reached from the given state s, and the set of states which can be reached from the given state s using transitions labeled with l in the transition set T, respectively:

- $out_T(s) = \{(s', l, s'') \mid (s', l, s'') \in T, s = s'\}$
- $reachable_T(s) = \{s_1, .., s_m\} \cup reachable_T(s_1) \cup \ldots \cup reachable_T(s_m)$ where $\{s_1, .., s_m\} = \{s'' \mid (s', l, s'') \in T,\ s = s'\}$
- $reachable_{T,l}(s) = \{s_1, .., s_m\} \cup reachable_{T,l}(s_1) \cup \ldots \cup reachable_{T,l}(s_m)$ where $\{s_1, .., s_m\} = \{s'' \mid (s', l', s'') \in T,\ s = s',\ l = l'\}$

Comparison of labels is useful to check whether exchanged messages are compatible or not. Formally, we define label compatibility as follows:

Definition 3.1 [Label compatibility] A label l_1 is compatible with a label l_2 if:

- $l_1 = (m_1, d_1, (p_{11}, \ldots, p_{1n}))$, $l_2 = (m_2, d_2, (p_{21}, \ldots, p_{2m}))$ such that $m_1 = m_2$, $d_1 = \overline{d_2}$, $n = m$, and $\forall\, k = 1, .., n \quad type(p_{1k}) = type(p_{2k})$ or,
- $l_1 = \tau$ and $l_2 = \tau$.

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

3.2 Successor States

Given two STSs $STS_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, 2\}$, we define a global state as a pair of states (s_1, s_2) where s_i is a current state of STS_i. Then, the protocol compatibility will be checked for all global reachable states. Therefore, we use the auxiliary functions $successors_{TS}((s_1, s_2))$ and $reachable_{TS}((s_1, s_2))$ which respectively provide the set of global states that can be reached, in one step, from a current global state (s_1, s_2), and the set of all global reachable states for two STSs, where TS stands for the considered τ handling strategy, namely strong, weak or trace, etc. Note that both $successors_{TS}((s_1, s_2))$ and $reachable_{TS}((s_1, s_2))$ sets are obtained by application of the COM and INE$_\tau$ rules given in Figure 2.

In our approach, the successor states function depends on the different ways of dealing with τ actions. In the following, we present the successor states function for the strong, weak and trace strategies.

We start by defining the *strong successor states*, in which every label held by a transition going out from a state of one service must be matched with a label hold by a transition going out from a state of the other service, even for τ actions.

Definition 3.2 [Strong Successor States] The set of global states which are *strongly* reachable, in one step, from a current global state (s_1, s_2) is defined as follows:

$$successors_{strong}((s_1, s_2)) = \{(s_1', s_2') \mid (s_1, l_1, s_1') \in T_1,\ (s_2, l_2, s_2') \in T_2 \text{ such that } l_1 \text{ and } l_2 \text{ are compatible}\}$$

We define now the *weak successor states*. As far as τ actions are concerned, we require that each τ action is matched by zero or more τ actions.

Definition 3.3 [Weak Successor States] The set of global states which are *weakly* reachable, in one step, from a current global state state (s_1, s_2) is defined as follows:

$$\begin{aligned} successors_{weak}((s_1, s_2)) = {} & \{(s_1', s_2') \mid (s_1, l_1, s_1') \in T_1,\ (s_2, l_2, s_2') \in T_2,\ l_1 \neq \tau \text{ such} \\ & \text{that } l_1 \text{ and } l_2 \text{ are compatible}\} \\ & \cup \{(s_1', s_2) \mid s_1' \in reachable_{T_1,\tau}(s_1), \textit{ and } reachable_{T_1,\tau}(s_1') = \\ & \varnothing \textit{ or } \exists (s_1', l_1, s_1'') \in T_1 \textit{ such that } l_1 \neq \tau\} \\ & \cup \{(s_1, s_2') \mid s_2' \in reachable_{T_2,\tau}(s_2), \textit{ and } reachable_{T_2,\tau}(s_2') = \\ & \varnothing \textit{ or } \exists (s_2', l_2, s_2'') \in T_2 \textit{ such that } l_2 \neq \tau\} \end{aligned}$$

A weaker definition of successor states consists in determining the set of global states that can be reached from the current one, in one step, by execution of the same traces. By doing so, τ actions are just not considered in the service protocols.

Definition 3.4 [Trace Successor States] The set of global states which are *trace* reachable, in one step, from a current global state (s_1, s_2) is defined as follows:

$$\begin{aligned} successors_{trace}((s_1, s_2)) = {} & \{(s_1', s_2') \mid (s_1, l_1, s_1') \in T_1, (s_2, l_2, s_2') \in T_2,\ l_1 \neq \tau \text{ and such} \\ & \text{that } l_1 \text{ and } l_2 \text{ are compatible}\} \\ & \cup \bigcup_{(s_1, \tau, s_1'') \in T_1} successors_{trace}((s_1'', s_2)) \\ & \cup \bigcup_{(s_2, \tau, s_2'') \in T_2} successors_{trace}((s_1, s_2'')) \end{aligned}$$

Example. We focus on small parts of our running example to illustrate how the successor states can be obtained for the strong, weak and trace strategies. Let us consider the global state (u_1, s_1):

$$\begin{aligned} successors_{strong}((u_1, s_1)) &= successors_{weak}((u_1, s_1)) \\ &= successors_{trace}((u_1, s_1)) \end{aligned}$$

$$= \{(u_2, s_2)\}$$

Since there is no τ action hold by a transition going out from u_1 and s_1, we obtain the same set of global states reachable, in one step, from (u_1, s_1) according to any τ strategy.

Now, let us consider the global state (u_4, s_4):

$successors_{strong}((u_4, s_4)) = \varnothing$

$successors_{weak}((u_4, s_4)) = \{(u_2, s_4), (u_5, s_4), (u_7, s_4)\}$

$successors_{trace}((u_4, s_4)) = \{(u_3, s_3), (u_6, s_5), (u_8, s_9)\}$

As it can be observed in Figure 3, all the transitions going out from u_4 hold τ actions, namely $\{((u_4, \tau, u_2)), (u_4, \tau, u_5), (u_4, \tau, u_7)\}$, while all actions hold by the transitions outgoing from s_4 are different to τ, namely $\{(s_4, searchFlight\ ?\ dis : string, dep : tdate, ret : tdate, s_3), (s_4, searchHotel\ ?\ dep : tdate, ret : tdate, cat : string, s_9), (s_4, searchCar\ ?\ d1 : date, d2 : date, type : string, s_5)\}$. By application of the *strong successor states* definition, the τ actions at u_4 do not match with any action at s_4, and none of the expected receptions at s_4 can be emitted at u_4. Hence, there is no global state that can be strongly reachable from (u_4, s_4). Considering the definition of *weak successor states*, the τ actions at state u_4 are skipped because all the transitions going out from this state are labeled with τ. Therefore, the set of global states weakly reachable, in one step, from (u_4, s_4) is $\{(u_2, s_4), (u_5, s_4), (u_7, s_4)\}$ since $reachable_{T_{user},\tau}(u_2) = \varnothing$, $reachable_{T_{user},\tau}(u_5) = \varnothing$ and $reachable_{T_{user},\tau}(u_7) = \varnothing$. Finally, following the *trace successor states* definition, as it is shown in Figure 3, the set of global states $\{(u_3, s_3), (u_6, s_5), (u_8, s_9)\}$ can be reached, in one step, from (u_4, s_4) using traces of compatible labels.

In order to obtain the set of all global reachable states for two STSs starting from an initial global state (s_1, s_2), we present the following definition:

$$reachable_{TS}((s_1, s_2)) = \{(s_1, s_2)\} \cup \bigcup_{(s_1', s_2') \in successors_{TS}((s_1, s_2))} reachable_{TS}((s_1', s_2'))$$

3.3 Transition Compatibility

As successor states, compatibility of transitions going out from a global state (s_1, s_2) depends on how τ actions are handled. Transition compatibility consists in checking if the sent (respectively, received) messages at a state s_1 could be received (respectively, sent) at another state s_2 and vice versa. To do so, we took inspiration from the equivalence notions stated by Milner [21] to present the following definitions:

Definition 3.5 [Strong Transition Compatibility] Given a global state (s_1, s_2), $comp\text{-}em\text{-}rec_{strong}((s_1, s_2)) = true$ iff $\forall (s_i, l_i, s_i') \in T_i$, with $i \in \{1, 2\}$, $\exists (s_j, l_j, s_j') \in T_j$, with $j = i\ mod\ 2\ + 1$, such that l_i and l_j are compatible.

Definition 3.6 [Weak Transition Compatibility] Given a global state (s_1, s_2), $comp\text{-}em\text{-}rec_{weak}((s_1, s_2)) = true$ iff:

- $\forall(s_i, l_i, s_i') \in T_i$ with $i \in \{1,2\}$, $l_i \neq \tau$, $\exists(s_j, l_j, s_j') \in T_j$, with $j = i \ mod\ 2 + 1$ such that l_i and l_j are compatible, and
- $\forall(s_i, \tau, s_i'') \in T_i$, with $i \in \{1,2\}$, $comp\text{-}em\text{-}rec_{weak}((s_i'', s_j)) = true$, and $\forall(s_j, \tau, s_j'') \in T_j$, with $j = i \ mod\ 2 + 1$, $comp\text{-}em\text{-}rec_{weak}((s_i, s_j'')) = true$.

Definition 3.7 [Trace Transition Compatibility] Given a global state (s_1, s_2), $comp\text{-}em\text{-}rec_{trace}((s_1, s_2)) = true$ iff:

- $\forall(s_i, l_i, s_i') \in T_i$, with $i \in \{1,2\}$, $l_i \neq \tau$, $\exists(s_j, l_j, s_j') \in T_j$, with $j = i \ mod\ 2 + 1$ such that l_i and l_j are compatible, and
- $\forall(s_i, \tau, s_i'') \in T_i$, *with* $i \in \{1,2\}$, $\exists(s, l, s') \in T_i$, $l \neq \tau$, such that $s \in reachable_{T_i,\tau}(s_i)$ then $\exists(s_i', s_j') \in successors_{trace}((s_1, s_2))$ where $s_i' = s'$.

Example. Figure 4 shows a part of our running example. Considering the global state (u_4, s_4), $com\text{-}em\text{-}rec_{TS}((u_4, s_4))$ returns *false* if *TS* is *strong* or *weak*, and *true* if *TS* is *trace*. Since there are transition labels at u_4 (respectively at s_4) which do not match with transition labels at s_4 (respectively at u_4), $com\text{-}em\text{-}rec_{strong}((u_4, s_4)) = false$. By application of the *weak transition compatibility* notion, $com\text{-}em\text{-}rec_{weak}((u_4, s_4)) = true$ iff:
$com\text{-}em\text{-}rec_{weak}((u_2, s_4)) = true$ and $com\text{-}em\text{-}rec_{weak}((u_5, s_4)) = true$ and $com\text{-}em\text{-}rec_{weak}((u_7, s_4)) = true$.
Let us focus on (u_2, s_4). The transition label $(searchCar\ ?\ d1 : date, d2 : date, type : string)$ at s_4 does not match with any transition label at u_2, and thus $com\text{-}em\text{-}rec_{weak}((u_2, s_4)) = false$. Therefore, $com\text{-}em\text{-}rec_{weak}((u_4, s_4)) = false$.
Last, by skipping the transitions holding τ actions at u_4, we always have traces of compatible labels starting from (u_4, s_4). Hence, $com\text{-}em\text{-}rec_{trace}((u_4, s_4)) = true$.

3.4 Formalising Some Service Compatibility Notions

Various notions of service compatibility exist [4,17,14,1,5] setting different requirements on how services are compatible. There is no standard or unique service compatibility notion, all of them are useful for different application domains. For illustration purposes, we focus here on three widely used notions, namely opposite behaviours, unspecified receptions and deadlock freeness.

Opposite behaviours. The strongest and most intuitive compatibility notion is that of *opposite behaviours*. In this paper, our definition of opposite behaviours compatibility is inspired from the bisimilarity concept originally stated by Milner [21]. Here, the opposite behaviours compatibility consists in the fact that when one service can send a message, then the other service must be willing to receive that message, and when one service is waiting to receive a message, then the other service must be sending that message. Hence, for any τ handling strategy *TS*, we formalise the *opposite behaviours* compatibility notion as follows:
Two STSs $STS_i = (A_i, S_i, I_i, F_i, T_i)$, $i = \{1,2\}$, are opposite behaviours compatible if and only if for each global state $(s_1, s_2) \in reachable_{TS}((I_1, I_2))$:

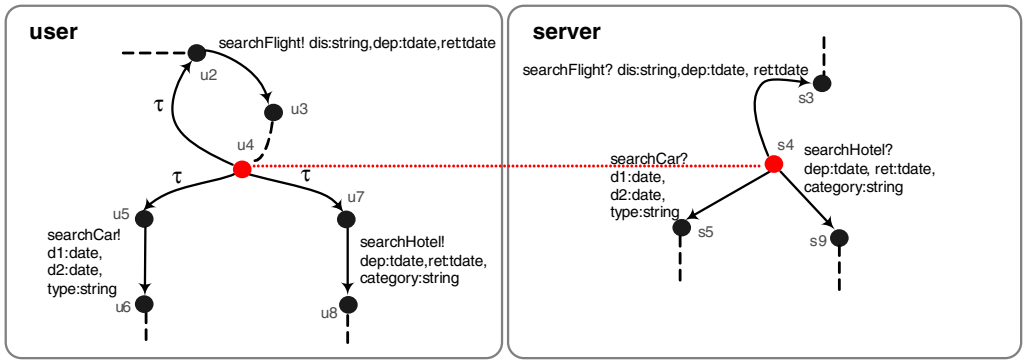

Fig. 4. Strong and Weak Incompatible Emissions Receptions at (u_4, s_4)

- $comp\text{-}em\text{-}rec_{TS}((s_1, s_2)) = true$, and
- if $s_1 \in F_1$ ($s_2 \in F_2$, respectively) then $s_2 \in F_2$ ($s_1 \in F_1$, respectively).

Example. Considering our running example, as we have explained in the previous section, $com\text{-}em\text{-}rec_{TS}((u_4, s_4)) = false$ if TS is *strong* or *weak*. Then, the user and server protocols are not opposite behaviours compatible with respect to the strong and weak strategies. However, these protocols are opposite behaviours compatible according to the trace strategy. For the user and server protocols, we can check that for each trace global reachable state (u_i, s_j) from (u_1, s_1) $com\text{-}em\text{-}rec_{trace}((u_i, s_j)) = true$. We can easily also check the second condition given in the *opposite behaviours* definition.

Unspecified receptions. One compatibility notion which is less restrictive than *opposite behaviours* is that of *unspecified receptions*. Here, our unspecified receptions definition is close to that given in [3,23] which consider that two service protocols are compatible even if one service can receive a message that cannot be sent by the other service, *i.e.* additional receptions, and this works in both directions. However, this definition requires that if one service can send a message at a reachable state, then that emission must be willing to be received by the other service. By doing so, it is also possible that one protocol holds an emission that will not be received by its partner as long as the state from which this emission goes out is unreachable during an established interaction.
Two STSs $STS_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in \{1, 2\}$, are unspecified receptions compatible if and only if for each global state $(s_1, s_2) \in reachable_{TS}((I_1, I_2))$:

- $comp\text{-}em\text{-}rec_{TS}((s_1, s_2)) = true$, and if $s_1 \in F_1$ ($s_2 \in F_2$, respectively) then $s_2 \in F_2$ ($s_1 \in F_1$, respectively), or
- $\exists (s_i, l_i, s_i') \in T_i$, *with* $i = \{1, 2\}$, $l_i = (m_i, ?, pl)$, $\nexists (s_j, l_j, s_j') \in T_j$, with $j = i \ mod \ 2 \ + 1$, such that l_i and l_j are compatible.

Example. In our travel advice running example, the user and server protocols are unspecified receptions incompatible according to the strong strategy since at the global state (u_4, s_4) there are transition labels, different to receptions, at u_4 (respectively s_4) which do not match with transition labels at s_4 (respectively u_4).

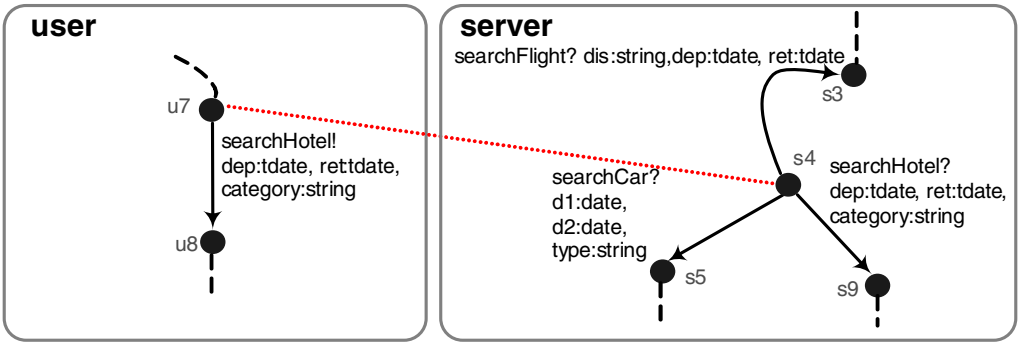

Fig. 5. Additional Receptions at (u_4, s_4)

However, both protocols are unspecified receptions compatible with respect to weak and trace strategies because for all global reachable states (u_i, s_j), if there is a transition label l at u_i (respectively s_j) which cannot be matched with a transition label at s_j (respectively u_i) thus l is a reception. For instance, let us consider the global reachable state (u_7, s_4) in Figure 5. At state s_4, there are two additional receptions, namely $(searchCar?d1 : date, d2 : date, type : string)$ and $(searchFlight?dis : string, dep : tdate, ret : tdate)$, that do not match with the emission at u_7.

Deadlock freeness. This notion is less restrictive than the two previous ones. It considers that two service protocols are compatible if and only if, starting from their initial states, they can always evolve until reaching a final global state. In order to formalise our deadlock freeness compatibility, we took inspiration from [2].
Two STSs $STS_i = (A_i, S_i, I_i, F_i, T_i)$, $i = \{1, 2\}$, are deadlock freeness compatible if and only if for each global state $(s_1, s_2) \in reachable_{TS}((I_1, I_2))$:

- $(s_1, s_2) \in (F_1 \times F_2)$, or
- $successors_{TS}((s_1, s_2)) \neq \varnothing$.

Example. Going back to the travel advice system, the user and server are incompatible with respect to the strong strategy since there exists no global strongly reachable state from (u_4, s_4), and u_4 and s_4 are not final states. Nevertheless, these two protocols are deadlock free compatible according to the weak and trace strategies because, starting from their initial states, for each weak (or trace) global reachable state (u_i, s_i) $successors_{TS}((u_i, s_i)) \neq \varnothing$ until reaching a final global state, where TS is *weak* or *trace*.

3.5 Compatibility Checking

We now present a protocol compatibility definition *UPC* for two given protocols described using STSs, a compatibility notion *CN* and a τ handling strategy *TS*. This definition enables one to check if two service protocols, described using STSs, are compatible with respect to *CN* and *TS*, and reports a counterexample *CE* if they are not.

$$UPC : \{STS_1, STS_2\} \times CN \times TS \rightarrow < Boolean, CE >$$

4 Encoding into Maude and Tool Support

In this Section, we respectively present a short introduction to Maude, the Maude encoding of our compatibility checking framework, and the prototype tool that automates the compatibility verification.

4.1 Maude Overview

Maude [6,8,7] is a high-level language and a high-performance interpreter and compiler in the OBJ [13] algebraic specification family that supports membership equational logic and rewriting logic specification and programming of systems. Thus, Maude integrates an equational style of functional programming with rewriting logic computation. In this section we provide an informal description of those Maude features necessary for understanding the paper; the interested reader is referred to [8,7] for more details.

Maude's underlying equational logic is membership equational logic [18], a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains.

In Maude, there are three different types of modules: functional, system and object-oriented modules. We focus here on functional modules (see [8,7] for additional information). For example, the following Maude functional module defines the natural numbers (with sorts `Nat` of natural numbers and `NzNat` of nonzero natural numbers), using the Peano notation, with the zero (`0`) and successor (`s_`) operators as constructors (note the `ctor` attribute). Then, the addition operation (`_+_`) is defined, being its behaviour specified by two equational axioms. The operators `s_` and `_+_` are defined using *mixfix* syntax (underscores indicate places for arguments).

```
fmod MY-NAT is
  sort NzNat Nat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [assoc comm id: 0] .
  vars M N : Nat .
  eq s M + s N = s (M + N) .
endfm
```

If an equational specification is confluent, terminating, and sort-decreasing, then it can be executed. Computation in a functional module is accomplished by using the equations as simplification rules from left to right until a canonical form is reached. Some equations, like the one expressing the commutativity of a binary operator, are not terminating, but nonetheless they are supported by means of *operator attributes*, so that Maude performs simplification *modulo* the equational theories provided by such attributes, that can be associative (`assoc`), commutativity

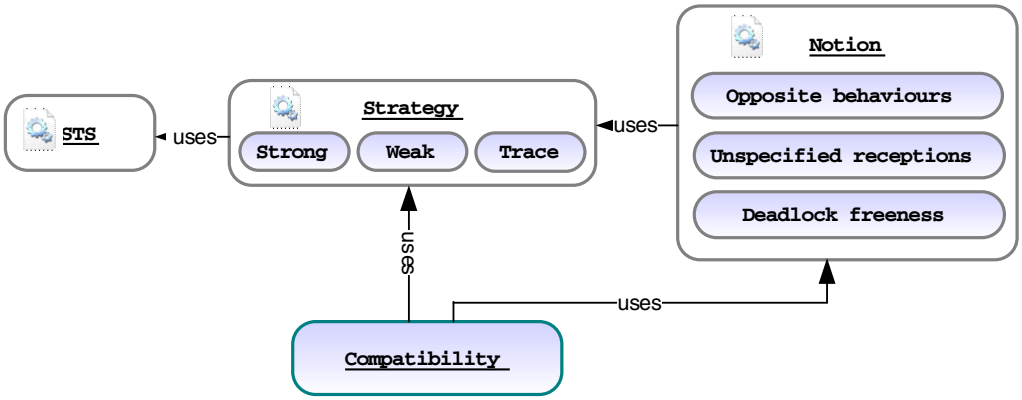

Fig. 6. Diagram of Maude Modules

(`comm`), and identity (`id`). The above properties must therefore be understood in the more general context of simplification *modulo* such equational theories.

In Maude, specifications may be generic, that is, they may be defined with other specifications as parameters. The requirements that a datatype must satisfy are described by *theories*. Parameterized modules are instantiated by means of *views*. A view shows how a particular module satisfies a theory, by mapping sorts and operations in the theory to sorts and operations in the target module, in such a way that the induced axioms are provable in the target module.

4.2 Maude Encoding

In this section, we will overview the different Maude modules (see Figure 6) used to encode our approach:

STS Specification. A module called `STS` specifies the service model. STSs are represented as objects of the form:

```
< Oid : STS | is : State, fss : Set{State}, ts : Set{Transition} >.
```

An *STS* object has an identifier (`Oid`), a type (`STS`), and an attribute set which consists of an initial state (`is`), a set of final states (`fss`), and a set of transitions (`ts`). A transition is a tuple of the form `St1 - L -> St2` where `St1` and `St2` are, respectively, the source and target states, and `L` is the label. A label is either a τ (`T`) or a tuple of the form `M d (PRS)`, where `M` is the message name, `d` stands for the direction being `?` or `!`, and `PRS` denotes the parameter list.

Tau Strategies. Two operations `successors` and `comp-em-rec` encode the $successors_{TS}$ and $comp - em - rec_{TS}$ functions, respectively, in three Maude modules called `STRONG`, `WEAK` and `TRACE`, respectively. For space reasons, we cannot give all the Maude specifications implementing these functions. For instance, Figure 7 shows the encoding of the $successors_{trace}$ function. The `successors` operation explores two STS objects (first and third arguments in line 1) starting from a global state (second and forth arguments in line 1) in order to return the set of global

```
1  op successors : Object State Object State -> Set{Pair{State,State}} .
2
3  ceq successors(< O  : sts  | (ts : St1 - M ! (PRS1) -> St1', TS ), AtS  >, St1,
4                  < O' : sts' | (ts : St2 - M ? (PRS2) -> St2', TS'), AtS' >, St2)
5     = (< St1', St2' >, successors(< O  : sts  | ts : TS, AtS  >, St1,
6                                   < O' : sts' | ts : TS', AtS' >, St2))
7       if compatible-parameters(PRS1, PRS2).
8
9  ceq successors(< O  : sts  | ts : (St1 - M ? (PRS1) -> St1', TS ), AtS  >, St1,
10                 < O' : sts' | ts : (St2 - M ! (PRS2) -> St2', TS'), AtS' >, St2)
11    = (< St1', St2' >, successors(< O  : sts  | ts : TS , AtS  >, St1,
12                                  < O' : sts' | ts : TS', AtS' >, St2))
13      if compatible-parameters(PRS1, PRS2) .
14
15 eq successors(< O  : sts  | ts : (St1 - T -> St1', TS), AtS  >, St1,
16                < O' : sts' | ts : TS', AtS' >, St2)
17    = (successors(< O  : sts  | ts : TS , AtS  >, St1',
18                  < O' : sts' | ts : TS', AtS' >, St2),
19       successors(< O  : sts  | ts : TS , AtS  >, St1,
20                  < O' : sts' | ts : TS', AtS' >, St2)) .
21
22 eq successors(< O  : sts  | ts : TS, AtS  >, St1,
23                < O' : sts' | ts : (St2 - T -> St2', TS'), AtS' >, St2)
24    = (successors(< O  : sts  | ts : TS, AtS  >, St1,
25                  < O' : sts' | ts : TS', AtS' >, St2'),
26       successors(< O  : sts  | ts : TS , AtS  >, St1,
27                  < O' : sts' | ts : TS', AtS' >, St2)) .
28
29 eq successors(STS1, St1, STS2, St2) = empty [owise] .
```

Fig. 7. Maude Encoding of Successors Function wrt the Trace Strategy

states (`Set{Pair{State,State}}`) reachable, in one step, from the current one using traces of compatible labels. Regarding the label compatibility, the expressions `St1 - M d1 (PRS1) -> St1'` and `St2 - M d2 (PRS2) -> St2'` in lines 8-9 require that the two transition labels at $St1$ and $St2$ have the same name and opposite directions such that $d_1 = \overline{d_2}$. Then, the `compatible-parameters(PRS1, PRS2)` call checks if both parameter lists `PRS1` and `PRS2` have identical type lists (same types in the same order).

Compatibility Notions. The opposite behaviours, unspecified receptions and deadlock freeness compatibility notions presented in Section 3.4 are specified into three operations `opposite-behaviours`, `unspecified-receptions` and `deadlock-freeness`, respectively. Each of these operations evaluates the compatibility of two STSs at all reachable global states with respect to a τ strategy. For instance, Figure 8 shows the specification of the *opposite behaviours* compatibility notion. In the very first step, the `opposite-behaviours` operation (line 1) accepts as arguments two STS objects and returns a tuple consisting of a Boolean value and a counterexample (`Tuple{Bool, Tuple{List{Transition}}}`). Then, in order to check the protocol compatibility at all global reachable states using a recursive execution, an auxiliary `opposite-behaviours` operation (lines 8-10) is called. As it is shown in line 13, the auxiliary `opposite-behaviours` operation calls the `comp-em-rec` operation for each global reachable state $(St1, St2)$. Depending on the returned Boolean value, it might be possible to recursively call (lines 14-15) the `opposite-behaviours` operation for each global state in SPS, *i.e.*, the set of global states that can be reached from $(St1, St2)$. In the case in which the `comp-em-rec` operation returns *false*, the auxiliary `get-counterexample` operation returns a couple of transition sequences starting from the initial states until reaching the incompatibility source. The recursive call can stop if the current global

```
1  op opposite-behaviours : Object Object -> Tuple{Bool, Tuple{List{Transition}}} .
2
3  op opposite-behaviours : Object Object Set{Pair{State,State}} Set{Pair{State,State}}
4                           -> Tuple{Bool, Tuple{List{Transition}}} .
5
6  eq opposite-behaviours(< O  : sts  | is : St1, AtS  >,
7                         < O' : sts' | is : St2, AtS' >)
8     = opposite-behaviours(< O  : sts  | is : St1, AtS  >,
9                           < O' : sts' | is : St2, AtS' >,
10                          < St1, St2 >, empty) .
11
12 ceq opposite-behaviours(STS1, STS2, (< St1, St2 >, SPS), SPS')
13    = if comp-em-rec(STS1, St1, STS2, St2)
14      then opposite-behaviours(STS1, STS2, (SPS, successors(STS1, St1, STS2, St2)),
15                               (< St1, St2 >, SPS'))
16      else < false, get-counterexample(STS1, STS2, < St1, St2 >) fi
17      if not < St1, St2 > in SPS' .
18
19 eq opposite-behaviours(STS1, STS2, SPS, SPS') = < true, empty > [owise] .
```

Fig. 8. Maude Encoding of Opposite Behaviour Notion

reachable state $(St1, St2)$ has previously been visited (line 17) in order to avoid an infinite execution in the case of looping protocols.

Compatibility Checking. A module `COMPATIBILITY{TSNOTION}` implements the compatibility definition. The parameter `TSNOTION` refers to a compatibility notion (opposite behaviours, unspecified receptions or deadlock freeness) encoded into a module called `NOTION{STRATEGY}`, according to a τ strategy (strong, weak or trace) taken as a parameter.

4.3 Tool Support

The compatibility checking framework that we have presented in the previous sections has been fully implemented into a prototype tool (see Figure 9 for an overview). The STS Maude specifications are automatically generated using our script STS2Maude implemented in Python. In order to check the protocol compatibility, we call the Maude's red(uce) command for the operation which implements the chosen compatibility notion (*e.g.*, `opposite-behaviours`), and the τ handling strategy. A Boolean value and a counterexample are returned indicating whether two STSs are compatible according to the checked compatibility notion and identifying the incompatibility source, respectively.

So far, our prototype tool has been validated on more than 75 examples, which range from small examples to real-world ones. The largest example of our database consists of two services where each one consists of more than 85 states and around 90 transitions. The maximum computation time for this example was about 7 seconds, and that was needed to check the trace deadlock freeness compatibility.

5 Related Work

During the last 25 years, compatibility has been intensively studied as a main issue in Software Engineering. Let us survey some key related works in this area. [3,23] have formally defined a compatibility notion for software components using an automata-based formalism. Their compatibility notion consists of checking two protocol properties, namely unspecified receptions and deadlock. More recently, [1,5]

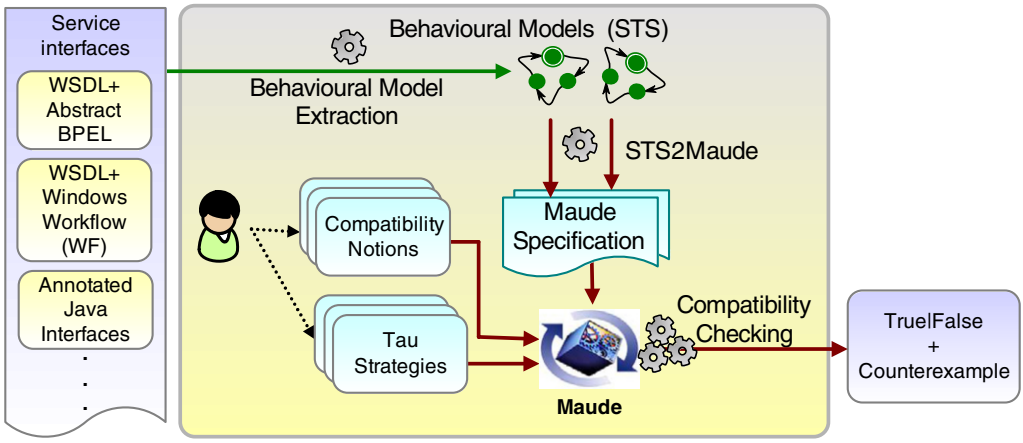

Fig. 9. Compatibility Checking Process

have used a Finite State Machine (FSM) model to formalise a compatibility notion for Web services which aims to check whether one service can substitute another service. In [1], the authors adopt an asymmetric relation, namely simulation, for determining if a new version of a service behaviour simulates a previous one. [5] gives a restrictive notion of behavioural compatibility saying that each trace in one Web service must also be preserved in another Web service. In [16,17,14], the authors rely on bisimulation algorithms to define the compatibility of Web services which are described using Petri nets. As regards process algebra, [4] and [10] have proposed a compatibility notion based on the π-calculus to ensure the successful composition of Software components and Web services, respectively. Bordeaux *et al.* [2] were the first who survey several compatibility notions. However, the Labeled Transition System (LTS) model presented in [2] does not consider value-passing and internal behaviour, and no tool support exists.

To sum up, there are few compatibility approaches [4,14,17] which take into account τ actions, and those that deal with them only use one of the possible strategies. Most of the previous works [3,5,2,1,17,14] consider models where value-passing is not taken into account. In addition, only a few proposals [1,5] have at their disposal an implementation to automatically check the proposed compatibility notions. Compared to these works, we consider a formal interface model which takes into account both non-observable actions and value-passing. Moreover, we propose a framework which enables to check, in a unified manner, several compatibility notions, and therefore make our proposal useful for many possible application domains. Our solution also deals with the subtleties of τ actions, and we have implemented several τ strategies to handle them properly. Since our proposal relies on an encoding into Maude, the compatibility check is fully automated. Last, the prototype tool that we implemented goes beyond the check of protocol compatibility since it is able to return a counterexample.

6 Concluding Remarks

In this paper we have presented a unified way to check different protocol compatibility notions using the rewriting logic system Maude. The compatibility notions presented in this paper deal with the subtlety of τ actions in service protocols. Our proposal is completely implemented into a prototype tool which enables a fully automated check of different compatibility notions with respect to the different τ handling strategies. Our prototype tool checks whether two protocols are compatible, but also returns a counterexample to identify the incompatibility source.

As regards future work, we first plan to extend our approach to check compatibility for n ($n > 2$) service protocols. We also aim at including more compatibility notions as well as more τ strategies, and to propose a methodology to make easier the extension of our approach with these new definitions. As far as possible applications are concerned, we would like to rely on our compatibility checking techniques in order to measure similarity of service protocols. Similarity goes farther than "Boolean" compatibility by detecting existing mismatches, and measuring the degree of compatibility of two (or more) protocols. Finally, we would like to propose a high-level language to enable a user to define his own compatibility notion, and some encoding techniques that would automatically generate the Maude code needed to verify this compatibility.

Acknowledgement

This work has been partially supported by the RESCUE (TIN2008-05932) and MDD-MERTS (TIN2008-03107) projects funded by the Spanish Ministry of Innovation and Science (MICINN).

References

[1] A. Aït-Bachir and M. Dumas and M.C. Fauvet. BESERIAL: Behavioural Service Interface Analyser. In *Proc. of the 6th International Conference on Business Process Management (BPM'08)*, volume 5240 of *Lecture Notes in Computer Science*, pages 374–377. Springer-Verlag, 2008.

[2] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of the 5th Int. Workshop on Technologies for E-Services (TES'04)*, volume 3324 of *Lecture Notes in Computer Science*, pages 15–28. Springer-Verlag, 2004.

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

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

[5] H.S. Chae, J.S. Lee, and J.H. Bae. An Approach to Checking Behavioral Compatibility between Web Services. *International Journal of Software Engineering and Knowledge Engineering*, 18(2):223–241, 2008.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.4 Manual, November 2008. Available in http://maude.cs.uiuc.edu.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C.L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[9] L. de Alfaro and T. Henzinger. Interface Automata. In *Proc. of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'01)*, pages 109–120. ACM Press, 2001.

[10] S.G. Deng, Z. Wu, M. Zhou, Y. Li, and J. Wu. Modeling Service Compatibility with Pi-Calculus for Choreography. In *Proc. of the 25th International Conference on Conceptual Modeling (ER'06)*, volume 4215 of *Lecture Notes in Computer Science*, pages 26–39. Springer-Verlag, 2006.

[11] H. Foster, S. Uchitel, and J. Kramer. LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and Choreography. In *Proc. of the 28th International Conference on Software Engineering (ICSE'06)*, pages 771–774. ACM Press, 2006.

[12] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. of the 13th International Conference on World Wide Web (WWW'04)*, pages 621–630. ACM Press, 2004.

[13] J. Goguen, T. Winkler, Jo. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

[14] N. Hameurlain. Flexible Behavioural Compatibility and Substitutability for Component Protocols: A Formal Specification. In *Proc. of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM'07)*, pages 391–400. IEEE Computer Society, 2007.

[15] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theor. Comput. Sci.*, 138(2):353–389, 1995.

[16] A. Martens. On Compatibility of Web Services. *Petri Net Newsletter*, 65:12–20, 2003.

[17] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing Compatibility of BPEL Processes. In *Proc. of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW'06)*, pages 147–156. IEEE Computer Society, 2006.

[18] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Proc. of the 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97)*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1997.

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

[20] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[21] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice-Hall, Inc., 1989.

[22] G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.

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