A generic framework for n-protocol compatibility checking

Francisco Durán, Meriem Ouederni, Gwen Salaün

A Generic Framework for N-Protocol Compatibility Checking

Francisco Durán[a], Meriem Ouederni[a], and Gwen Salaün[b]

[a] *Department of Computer Science, University of Málaga, Spain*
[b] *Grenoble INP–INRIA–LIG, France*

Abstract

Service-Oriented Computing promotes the development of new systems from existing services which are usually accessed through their public interfaces. In this context, interfaces must be *compatible* in order to avoid interoperability issues. In this article, we propose a new framework for checking the compatibility of n service interfaces. Our framework is *generic*, in the sense that it implements several compatibility notions useful for different application areas, and *extensible* since new further notions can easily be incorporated. We consider a service interface model which takes behavioural descriptions with value-passing and non-observable actions into account. Our compatibility checking framework has been fully implemented into a prototype tool which relies on the rewriting logic-based system Maude.

Key words: SOC, Transition Systems, Non-observable Actions, Compatibility, Maude.

1. Introduction

Service-Oriented Computing (SOC) aims at developing new software systems by reusing existing services. Services are software applications which are developed independently, loosely coupled, and accessed through their public interfaces. Thus, it is essential to ensure that service interfaces *fit* with each other in the system being developed. To this end, compatibility checking is of utmost importance to guarantee that interfaces are safely reused,

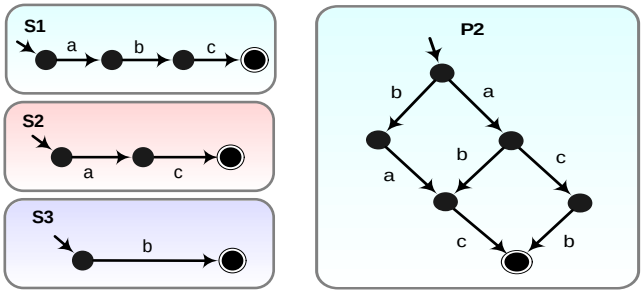
Email addresses: duran@lcc.uma.es (Francisco Durán), meriem@lcc.uma.es (Meriem Ouederni), Gwen.Salaun@inria.fr (and Gwen Salaün)

so that they can successfully interoperate. However, verifying compatibility is difficult, especially when the model of service interfaces takes interaction protocols (messages and their application order) into account, which allows one to avoid erroneous behaviours or deadlock situations when executing a set of services together [40].

There exists much work on the compatibility of services described using different interface models (see, for instance, [47, 6, 7, 15] for automata, [9, 16] for π-calculus, [24, 30] for Petri nets, and [3, 10] for state machines). Nevertheless, these proposals usually do not consider value-passing, *i.e.*, parameters coming with the messages exchanged between services, and non-observable or internal (τ) actions. Furthermore, most of the existing approaches on service compatibility rely on one particular compatibility notion defined with respect to a specific application domain (see, for instance, [30] for service composition, [3, 10, 24] for service substitution, or [16] for service choreography), and the compatibility is often checked for two services (very few contributions on n-services compatibility exist, see Section 5). Lastly, only a few existing proposals are equipped with appropriate tools for automating the service compatibility check.

To fill these gaps, we consider a generic approach for checking the compatibility of n ($n \geq 2$) service interfaces (signature and interaction protocols). The interaction protocols are described by means of *Symbolic Transition Systems* (STSs), and take value-passing as well as internal behaviours into account. We formalise several compatibility notions for n services, considering different strategies for handling value-passing. We propose a framework where these notions of compatibility can be automatically checked in a unified way. It is also possible to return a counter-example in order to detect the incompatibility source. This framework is supported by a Maude-based implementation and can be extended with new compatibility notions and new strategies to handle value-passing.

Although studying the compatibility of n interacting protocols is much harder when $n > 2$, it is easy to find situations in which checking the compatibility of n protocols is required, *e.g.*, in the composition of several services. In what follows, we introduce an example to illustrate that the compatibility of n services cannot always be computed using techniques existing for two protocols [17, 6]. Let us consider a system with three services, S1, S2 and S3, which interoperate through the exchange of messages a, b and c. Figure 1(a) shows the service behaviours described using transition systems (presented in Section 2). We assume a synchronous binary communication model and a



(a) Service behaviours. (b) Composition of S2 and S3.

Figure 1: N-service compatibility.

compatibility notion which requires that two services are compatible if they can always synchronise on each observable action (we refer to this notion as *bidirectional complementarity* or BC for short). A naive solution to check the compatibility of S1, S2 and S3 using two service-based techniques is as follows. We could first compose $n-1$ services (here, $n=3$) and then check the compatibility of the composite service with the remaining one. For instance, the composition of S2 and S3 returns the transition system P2 which consists of interleaved traces shown in Figure 1(b).[1] Here, P2 can start by performing either a or b, whereas S1 always performs a and then b. As a consequence, the protocols P2 and S1 are not BC compatible because they cannot synchronise on b at their initial states, *i.e.*, message b does not have any match. On the other hand, if we consider the three services all at once, their compatibility can be checked as follows: initially, S1 and S2 can synchronise on message a, S3 waits until it can perform b with S1, and lastly, the system terminates successfully once S1 and S2 synchronise on message c. Thus, the three services are compatible. This simple example illustrates the need to extend the existing compatibility notions and their checking from two to $n \geq 2$ services.

In this article, we illustrate our approach for checking n-service compatibility with some well-known compatibility notions, namely *bidirectional complementarity* [6] (BC), *unspecified receptions* [7, 47] (UR), *deadlock-freeness* [6] (DF), *one path* [16] (OP), and *unidirectional complementarity* (UC). Considering different compatibility notions makes our framework useful in different

[1] In this example, services are composed using a CCS-like semantics [35].

contexts, *e.g.*, in the case of the client/server model one would prefer the *UC* notion (further arguments are given throughout Section 3.3). Our proposal can also be used for several service issues, such as composition, adaptation, substitution, or discovery. For instance, in the context of service adaptation [32], a counter-example might be helpful to automate the specification of an intermediate service, *i.e.*, adaptor, which enables continuing the communication in spite of existing incompatibilities.

A prototype tool implementing our framework has been developed in the rewriting logic system Maude [11, 12]. Other alternatives were studied, such as using process algebraic tools (*e.g.*, CADP [22] or MCRL2 [23]), but we found the Maude system more convenient than these tools for the development of our generic framework encoding the different compatibility definitions. We present in Section 6 a short comparison between our approach and these alternative solutions.

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

2. Behavioural Service Model

This section presents the model of service interfaces we use for the compatibility checking, and relates it to existing models and platform languages. We assume that service interfaces are equipped both with a signature, *i.e.*, a set of profiles of required and provided operations together with their argument types, and an interaction protocol specifying the service behaviour. In our approach, protocols are represented by Symbolic Transition Systems (STSs). In an STS, communication between services is represented using *events* relative to the emission and reception of messages corresponding to operation calls. An event comes with a list of parameters (possibly empty) whose types respect the operation signature. A *label* describes either the (internal) τ action or an event using the tuple (m, d, pl) where m is the message name, d stands for the communication direction (either an emission ! or a reception ?), and pl is either a list of data terms if the label corresponds to an emission, or a list of variables if the label is a reception.

Definition 1 (STS)**.** *A Symbolic Transition System, or STS, is a tuple* (A, S, I, F, T) *where: A is an alphabet which corresponds to the set of la-*

bels associated to transitions, S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ is a nonempty set of final states, and $T \subseteq S \backslash F \times A \times S$ is the transition relation.

There are several *symbolic* transition systems in the literature, *e.g.*, [18, 27, 41, 25]. These models were proposed as solution to the state explosion problem, and extend Labelled Transition Systems with data types, value-passing events and guarded transitions. Our STS model is a variant of the STG (Symbolic Transition Graph) model presented in [25], where guards in branching transitions are abstracted into transitions labelled with τ actions. Keeping an explicit representation of guards in the STS definition would allow us to have an abstraction closer to the service implementation, as presented in [41, 18]. However, when checking compatibility at design-time, our goal is to ensure the correct service interaction at run-time. Since we do not know at design-time the values exchanged between services, guards cannot be evaluated. Therefore, having guards in service models does not help when verifying compatibility, and considering all possible evolutions (replacing guards with τ transitions) ensures (if services are compatible) a correct interaction whatever values are exchanged.

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 platform languages (*e.g.*, Abstract BPEL or WF). For instance, such abstractions for Web services were used in [20, 43, 19, 8] for verification, composition or adaptation purposes. We give a short discussion on this in Section 2.2.

For the sake of clarity, in the rest of the article, we will describe service interfaces only with their corresponding STSs. The signature will be left implicit, yet it can be inferred from the typing of arguments (made explicit here) in STS labels.

2.1. STS Operational Semantics

The operational semantics of one $STS = (A, S, I, F, T)$ (denoted by $\rightarrow_b$) is defined with three rules given in Figure 2 for, respectively, internal action (TAU), emission (EM), and reception (REC). In the semantic rules, for readability reasons, we consider either one output value or one input variable instead of using lists. This can be easily generalised to lists. The pair $\langle s, E \rangle$

consists of an active state[2] $s \in S$ and a data environment E. A data environment is a set of pairs $\langle x, v\rangle$ where x is a variable and v is a value. The type of x is returned using the function *type*. The environment can be updated using the operator "$\oslash$" which assigns a new value to an existing variable, or adds a new variable and its value to the environment:

$$E \oslash \langle x', v'\rangle \triangleq \begin{cases} E' \cup \{\langle x, v'\rangle\} & \text{if } \exists \langle x, v\rangle \in E \text{ such that } x = x' \text{ with} \\ & E' = E \setminus \{\langle x, v\rangle\} \\ E \cup \{\langle x', v'\rangle\} & \text{otherwise} \end{cases}$$

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

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

where the expression $E(x)$ returns the value of x in the environment.

Notice that, using the STS model, a choice can be represented using either a state and at least two outgoing transitions labelled with observable actions (external choice) or branches of τ actions (internal choice). The operational semantics of n STSs $STS_{i\in\{1\ldots n\}} = (A_i, S_i, I_i, F_i, T_i)$ (denoted by $\rightarrow_c$) is formalised using the synchronous communication[3] rule COM and the independent evolution rule INE_τ given in Figure 3, where $\{as_1, \ldots, as_n\}$ denotes the set of active states. In the COM rule,[4] value-passing and variable substitutions rely on a late binding semantics [36].

2.2. Internal Behaviours

Internal behaviours correspond to abstractions of pieces of code, *e.g.*, conditions involving variables and functions, that a service can perform independently from its partners. Service analysis can be worked out without taking into account these behaviours since they are non-observable from its partners' point of view. However, considering the non-observable behaviours

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

[3]Although checking protocol compatibility is undecidable with asynchronous communication [7], Fu *et al.* proved in [21] that a large class of interfaces can be analysed under an asynchronous communication model using existing techniques and tools for the synchronous communication model.

[4]In the operational semantics of n STSs we keep the resulting action observable (as it is the case in process algebras such as CSP and LOTOS).

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

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

$$\frac{(s \xrightarrow{a?x} s') \in T}{\langle s, E\rangle \xrightarrow{a?x}_b \langle s', E\rangle} \quad \text{(REC)}$$

Figure 2: Operational semantics of an STS.

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

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

Figure 3: Operational semantics of n STSs.

while analysing services helps us to find out possible interoperability issues. Indeed, although one service can behave as expected by its partner from an external point of view, interoperability issues may occur because of unexpected internal behaviours that services can execute. For instance, Figure 4 shows two versions of one service protocol with (STS2') and without (STS2) its internal behaviour. As we can see, STS2 and STS1 can perfectly interoperate under the synchronous semantics because each service can send (respectively receive) the messages expected (respectively sent) by its partner. However, if we consider STS2', which is an abstraction closer to what the service actually does, we see that this protocol can (choose to) execute a τ action at state s1 and arrive at state s3 while STS1 is still in state u1. At this point, STS2' and STS1 cannot exchange messages, and the system deadlocks. This issue would not have been detected with STS2.

Typically, higher-level languages, such as abstract BPEL or abstract Windows workflow (WF), are used in the literature (see, *e.g.*, [32, 14, 31]) to provide abstract descriptions (Interface Description Languages) of service behaviours. Here we focus on WF to illustrate how STSs, and in particular τ transitions, can be extracted from workflow-based notations. WF describes

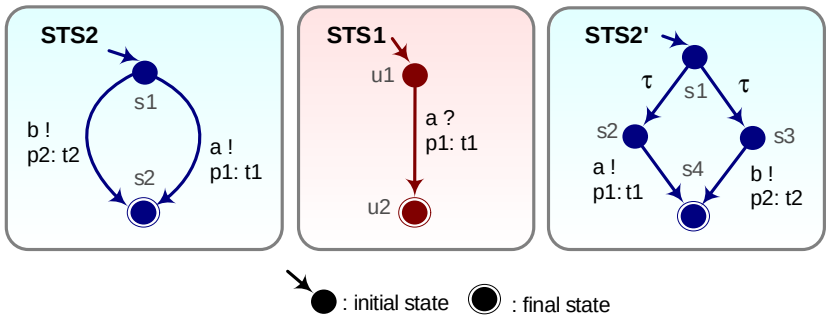


Figure 4: STS1 and STS2 interoperate successfully, but STS1 and STS2' can deadlock.

service behaviours using a set of basic activities, *e.g.*, IfElse, Listen and While, for which it is useful to keep some τ transitions in their respective STS descriptions.

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

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

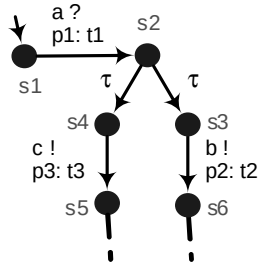
The While activity is used to repeat an activity as long as the loop condition is satisfied. Hence, the corresponding STS encodes this activity using a non-deterministic choice, specified using τ transitions, between the looping behaviour and the behaviour that can be executed after the While activity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

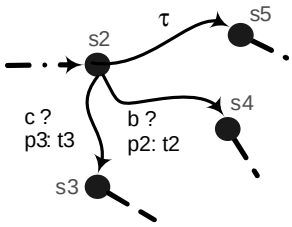
Other abstract WF activities such as Terminate, Parallel and Code can also generate τ transitions in the corresponding STS model.

3. *N*-Protocol Compatibility

In this section, we first introduce some preliminary definitions necessary for the formalisation of protocol compatibility. Then, we present the concept

Abstract WF activity	STS description

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

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

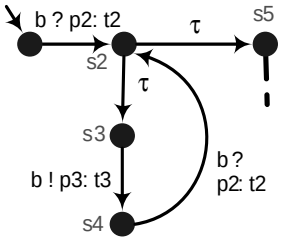
`WebServiceInput(b?(p2:t2))`
`While`
`(`
`(p2 < 10),`
`InvokeWebService`
`(b!(p3:t3), b?(p2:t2))`
`)`
`...`
	

Table 1: Examples of Abstract WF activities and their corresponding STSs.

of *state compatibility* on which some of the compatibility notions we formalise in this article rely.

3.1. Preliminaries

Given n services described using STSs $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, we define a global state as a tuple of states $(s_1, \ldots, s_n)$ where s_i is the state of STS_i. In what follows, we describe a transition using a tuple (s, l, s') such that s and s' denote the source and target states, respectively, and l stands for its label. Lastly, for the sake of clarity, we assume in the rest of this article that the functions we define have access to the STSs $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$ even if they are not explicitly passed as input parameters.

Label Compatibility. Label comparison is necessary to check whether exchanged messages (and their parameters) are compatible. However, data parameters coming with messages can be handled differently in order to check their compatibility. Thus, our n-protocol compatibility check is parameterised by a parameter-handling strategy (PS). In particular, we illustrate our approach by considering two of the possible ways of dealing with message parameters. The usual meaning of parameter compatibility requires that the parameter list expected to be received perfectly matches (same types in the same order) the parameter list coming with the sent message. We refer to this definition as *parameter matching*, *pm* for short (see Definition 2). Alternatively, if one service receives a message, then it can receive less parameters than those being sent. This definition is referred to as *unspecified parameters*, *up* for short (see Definition 3).

Definition 2 (Parameter Matching)**.** *Two parameter lists,* $pl_1 = (p_{11}, \ldots, p_{1n})$ *and* $pl_2 = (p_{21}, \ldots, p_{2m})$*, are pm-compatible,* $param\text{-}comp_{pm}(pl_1, pl_2)$*, if and only if:*

- $n = m$*, and*
- $\forall k \in \{1, \ldots, n\}$*,* $type(p_{1k}) = type(p_{2k})$*.*

Definition 3 (Unspecified Parameters)**.** *A parameter list* $pl_1 = (p_{11}, \ldots, p_{1n})$*, coming with an emission message, is up-compatible with a parameter list* $pl_2 = (p_{21}, \ldots, p_{2m})$*, coming with a reception message,* $param\text{-}comp_{up}(pl_1, pl_2)$ *if and only if:*

- $n \geq m$*, and*
- $\forall k \in \{1, \ldots, m\}$*,* $type(p_{1k}) = type(p_{2k})$*.*

Notice that the *up* parameter-handling strategy discards the last remaining parameters in the emission message. More sophisticated strategies in which, *e.g.*, the parameters can be sent and received in different order, or in which the types do not need to coincide (*e.g.*, subtyping or automatic conversion) may similarly be defined. Strategies in which, *e.g.*, we compare the semantics of parameter names and/or types using, for instance, the Wordnet similarity package [39], are also possible.

We now define label compatibility, which depends on the parameter-handling strategy PS, as follows:

Definition 4 (Label Compatibility)**.** *Given labels l_1 and l_2, lab-comp$_{PS}(l_1, l_2)$ if and only if:*

- $l_1 = (m_1, d_1, pl_1)$ *and* $l_2 = (m_2, d_2, pl_2)$, $m_1 = m_2$, $d_1 = \overline{d_2}$ *and param-comp$_{PS}(pl_1, pl_2)$, or*
- $l_1 = \tau$ *and* $l_2 = \tau$,

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

Other notions of label compatibility could similarly be used. For instance, message names can be compared using techniques such as structural comparison using the N-gram algorithm [28] or semantical comparison using the Wordnet similarity package.

Reachable States. In order to verify the service compatibility, we need to check every global state that can be reached during system execution. Therefore, we define a function *reachable$_{PS}((s_1, \ldots, s_n))$* which provides the set of global states that n interoperating services can reach, in one or more steps, from a current global state $(s_1, \ldots, s_n)$ through synchronisations or independent evolutions. In our definition of reachability, sequences of τ-actions are skipped. States reached with τ actions are in the set of reachable states only if they are final or if there are observable actions that can be executed from them. *PS* is the considered parameter-handling strategy, and the STSs are implicitly accessed as indicated above.

Definition 5 (Reachable States)**.** *Given a parameter-handling strategy PS and n STSs $STS_{i \in \{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, the set of global states reachable from a global state $(s_1, \ldots, s_n)$, reachable$_{PS}((s_1, \ldots, s_n))$, is the smallest set such that $\forall i \in \{1 \ldots n\}$, $\forall (s_i, l_i, s'_i) \in T_i$:*

- *if $l_i = \tau$, then*
  - *reachable$_{PS}((s_1, \ldots, s'_i, \ldots, s_n)) \subseteq$ reachable$_{PS}((s_1, \ldots, s_n))$, and*
  - *if $s'_i \in F_i$, or $\exists (s'_i, l'_i, s''_i) \in T_i$ such that $l'_i \neq \tau$, $(s_1, \ldots, s'_i, \ldots, s_n) \in$ reachable$_{PS}((s_1, \ldots, s_n))$.*
- *else, $\forall j \in \{1 \ldots n\}$, $j \neq i$, $\forall (s_j, l_j, s'_j) \in T_j$ with lab-comp$_{PS}(l_i, l_j)$, $\{(s_1, \ldots, s'_i, \ldots, s'_j, \ldots, s_n)\} \cup$ reachable$_{PS}((s_1, \ldots, s'_i, \ldots, s'_j, \ldots, s_n)) \subseteq$ reachable$_{PS}((s_1, \ldots, s_n))$.*

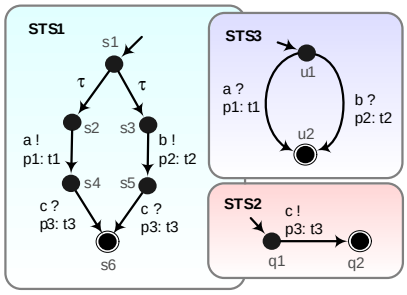


Figure 5: *STS*s of three interacting services.

Note that the resulting set is obtained by the application of the INE_τ and COM rules given in Figure 3 as follows. Firstly, if there is an internal transition (s_i, τ, s_i'), the application of the INE_τ rule implies the recursive reachability computation in $(s_1, \ldots, s_i', \ldots, s_n)$ due to this uncontrolled evolution. Furthermore, the global state $(s_1, \ldots, s_i', \ldots, s_n)$ must be added to the set of reachable global states if s_i' is a final state, *i.e.*, the respective service cannot evolve any more, or there exists at least one observable label going out from s_i'. Lastly, the application of the COM rule allows us to compute the global states reachable through possible synchronisations, where the labels are compared using the function $\mathit{lab\text{-}comp}_{PS}$.

Example 1. Let us consider the STSs given in Figure 5. STS1 can receive message c after sending either request a or b depending on an internal choice. STS2 can simply send message c. Lastly, STS3 can either wait for a reception of message a or b. The set of global states which can be reached from $(\mathsf{s1}, \mathsf{q1}, \mathsf{u1})$ for STS1, STS2, and STS3 is the following:

$$\begin{aligned}&\mathit{reachable}_{pm}((\mathsf{s1}, \mathsf{q1}, \mathsf{u1}))\\ &\quad = \{(\mathsf{s2}, \mathsf{q1}, \mathsf{u1}), (\mathsf{s4}, \mathsf{q1}, \mathsf{u2}), (\mathsf{s3}, \mathsf{q1}, \mathsf{u1}), (\mathsf{s5}, \mathsf{q1}, \mathsf{u2}), (\mathsf{s6}, \mathsf{q2}, \mathsf{u2})\}\end{aligned}$$

Deadlock Freeness. In order to check that services can always interoperate starting from a given global state, we define deadlock-freeness as follows:

Definition 6 (Deadlock Freeness)**.** *Given a parameter strategy PS and n STSs* $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$*, the set of their deadlock-free global states,* df_{PS}*, is the least set such that* $(s_1, \ldots, s_n) \in df_{PS}$ *if and only if either* $(s_1, \ldots, s_n) \in (F_1 \times \cdots \times F_n)$ *or* $\forall (s_1', \ldots, s_n') \in \mathit{reachable}_{PS}((s_1, \ldots, s_n))$*,* $(s_1', \ldots, s_n') \in df_{PS}$*.*

3.2. State Compatibility

Service interaction basically depends on synchronisations over observable actions and then can be defined using a criterion set on them. The criterion is used to check the state compatibility as follows. For a given global state $(s_1, \ldots, s_n)$, this state is considered compatible if each message sent (respectively, received) by one protocol i at state $s_i \in S_i$ will be eventually received (respectively, sent) by another protocol j at state $s_j \in S_j$ where $i, j \in \{1 \ldots n\}$ and $i \neq j$, such that all protocols evolve into a compatible global state. If there is no such protocol j able to interact with protocol i's action, protocols $\{1 \ldots n\}$ should be able to reach a global state in which this action will be enabled. Since services can evolve independently through some non-observable τ transitions, the behavioural compatibility requires that *each* internal evolution must lead all services into compatible states [9, 15]. Therefore, if there exist transitions $(s_i, \tau, s'_i) \in T_i$, the compatibility must be checked at the target states.

Definition 7 (State d-Compatibility)**.** *Given a parameter strategy PS, a label direction d, and n STSs $STS_{i \in \{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$, the set of (PS, d)-compatible states, state-comp$_{PS,d}$, is the largest set such that if a global state $(s_1, \ldots, s_n) \in$ state-comp$_{PS,d}$ then $\forall i, j \in \{1 \ldots n\}$, $i \neq j$, $\forall (s_i, l_i, s'_i) \in T_i$:*

- *if $l_i = (m_i, d, pl_i)$, then*
  - *$\exists (s_j, l_j, s'_j) \in T_j$ such that lab-comp$_{PS}(l_i, l_j)$ and $(s_1, \ldots, s'_i, \ldots, s'_j, \ldots, s_n) \in$ state-comp$_{PS,d}$ or*
  - *$\exists (s'_1, \ldots, s_i, \ldots, s'_j, \ldots, s'_n) \in$ reachable$_{PS}((s_1, \ldots, s_n))$ such that $\exists (s'_j, l'_j, s''_j) \in T_j$ where:*
    - *lab-comp$_{PS}(l_i, l'_j)$,*
    - *$(s'_1, \ldots, s_i, \ldots, s'_j, \ldots, s'_n) \in$ state-comp$_{PS,d}$, and*
    - *$(s'_1, \ldots, s'_i, \ldots, s''_j, \ldots, s'_n) \in$ state-comp$_{PS,d}$,*
- *else if $l_i = \tau$, then $(s_1, \ldots, s'_i, \ldots, s_n) \in$ state-comp$_{PS,d}$.*

Example 2. Let us consider again the STSs given in Figure 5. Since STS1 can internally transit to s2 or s3, $(\mathsf{s1}, \mathsf{q1}, \mathsf{u1}) \in$ *state-comp*$_{pm,?}$ only if $(\mathsf{s2}, \mathsf{q1}, \mathsf{u1})$ and $(\mathsf{s3}, \mathsf{q1}, \mathsf{u1})$ are in *state-comp*$_{pm,?}$. The check for the last two states shows that the receptions b and a do not have any match at the global

states $(\mathsf{s2}, \mathsf{q1}, \mathsf{u1})$ and $(\mathsf{s3}, \mathsf{q1}, \mathsf{u1})$, respectively. Hence, these states are not compatible, and as a consequence the initial state $(\mathsf{s1}, \mathsf{q1}, \mathsf{u1})$ is also non compatible.

3.3. Service Compatibility Notions

In this article, we focus on compatibility notions which do not require human intervention to set the particular properties of the services involved.[5] We illustrate our approach by considering some compatibility notions that exist in the literature for analysing two protocols, namely *BC*, *UR*, *DF*, *OP*, and *UC*. We extend these definitions to n protocols described using our STS model. In addition, the compatibility notions formalised here can be checked with respect to the different strategies for handling parameters (*PS*) that we have presented above.

Bidirectional Complementarity (*BC*). The most intuitive notion of compatibility is possibly that of *BC*. This notion requires that when one service can send a message, there is another service which *eventually* receives that message, and when one service is waiting to receive a message, then there is another service which must *eventually* send that message. Furthermore, the protocols must be deadlock-free.

Definition 8 (Bidirectional Complementarity Compatibility)**.** *Given a parameter strategy PS, n STSs* $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$ *are BC compatible if and only if:*

- $(I_1, \ldots, I_n) \in$ *state-comp*$_{PS,?}$,
- $(I_1, \ldots, I_n) \in$ *state-comp*$_{PS,!}$, *and*
- $(I_1, \ldots, I_n) \in df_{PS}$.

Example 3. Figure 6 shows three STSs that are not compatible with respect to *BC*, and neither the *pm* nor the *up* parameter strategies. The incompatibility is detected at the global state $(\mathsf{s2}, \mathsf{q2}, \mathsf{u1})$, reachable from the initial one, since STS1, STS2 and STS3 are not able to reach any global state in which message d at state s2 can match.

[5] Checking of some compatibility notions cannot be fully automated because they need as input both the service interfaces and some properties that services must ensure. For

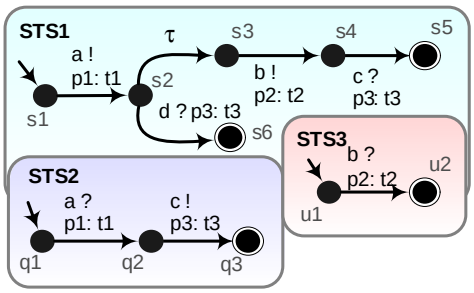


Figure 6: Bidirectional complementarity vs. unspecified receptions compatibility.

Unspecified Receptions (*UR*). The *BC* compatibility notion can be very restrictive in some situations. Imagine for instance a service able to receive all requests from a client, but which can also accept other receptions from other clients. In these cases, the *UR* compatibility notion may be more appropriate. The definition that we propose for this notion is inspired by those proposed in [7, 47].

The *UR* notion requires that if one service can send a message at a reachable state, then there is another service which must *eventually* receive that emission. Thus, n service protocols are compatible even if one service is able to receive a message that cannot be sent by any of the other services, *i.e.*, there might be additional receptions. By doing so, it is possible that one protocol holds an emission that will not be received by its partners as long as the state from which this emission goes out is unreachable when protocols are interacting together. Furthermore, the protocols must be deadlock-free.

Definition 9 (Unspecified Receptions Compatibility)**.** *Given a parameter strategy PS, n STSs* $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$ *are UR compatible if and only if:*

- $(I_1, \ldots, I_n) \in$ *state-comp*$_{PS,!}$ *and*
- $(I_1, \ldots, I_n) \in df_{PS}$.

Example 4. Going back to the STSs given in Figure 6, we note that they are *UR* compatible with respect to both *pm* and *up*. No incompatibility is

example, the notion of *goal-oriented* compatibility [48] aims at verifying some temporal properties, provided by the user, over protocol specifications using model-checking techniques.

detected since there is a unique unmatched message (d), at the reachable global state (s2, q2, u1), but this is a reception.

Deadlock Freeness (*DF*). The *DF* compatibility notion relaxes the strong requirements of the aforementioned notions. It considers that n service protocols are compatible if and only if, starting from their initial global state, they can always evolve until reaching a final global state.

Definition 10 (Deadlock Freeness Compatibility)**.** *Given a parameter strategy PS, n STSs* $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$ *are DF compatible if and only if the initial global state* $(I_1, \ldots, I_n) \in df_{PS}$.

Example 5. The STSs in Figure 7 are not *DF* compatible since, for the global state (s6, q2, u1) reachable from the initial one, (s6, q2, u1) $\notin df_{PS}$.

One path (*OP*). The weakest notion we present in this article is *OP* compatibility, introduced in [16] for the case of two services. N service protocols are *OP* compatible if and only if starting from their initial states, they can at least execute a sequence of interactions until reaching a final global state. Notice that this notion does not ensure deadlock-freeness of the whole system. For instance, if the path leading to a final global state involves an internal choice, it is not required that all branches in that choice lead to final global states. However, the *OP* compatibility has some interest for systems without uncontrolled internal choices. In this setting, the system execution can be controlled (using, *e.g.*, controller synthesis techniques [49, 42]) to enforce the execution of a specific path.

Definition 11 (One Path Compatibility)**.** *Given a parameter-handling strategy PS, n STSs* $STS_{i\in\{1,\ldots,n\}} = (A_i, S_i, I_i, F_i, T_i)$ *are OP compatible if and only if there exists a global state* $(s_1, \ldots, s_n) \in reachable_{PS}((I_1, \ldots, I_n))$ *such that* $(s_1, \ldots, s_n) \in (F_1 \times \cdots \times F_n)$.

Example 6. Let us consider again the STSs in Figure 7. They are *OP* compatible because the final global state (s5, q3, u2) $\in reachable_{pm}$((s1, q1, u1)).

Unidirectional Complementarity (*UC*). So far, we have presented a set of symmetric notions for checking service compatibility. Now, we present an asymmetric compatibility notion, namely *UC*. Two services are *UC* compatible if and only if there is one (complementer) service which is able to

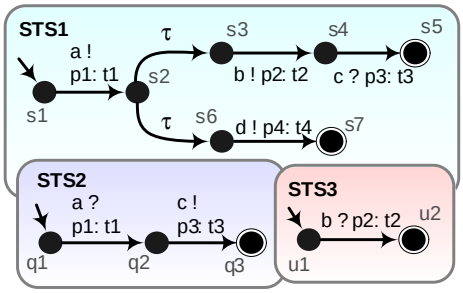


Figure 7: Deadlock freeness vs. one path compatibility.

eventually receive (respectively, send) all messages that its (complemented) partner expects to send (respectively, receive) at all global reachable states. Intuitively, the complementer service may send and receive more messages than the complemented service.

In order to extend the complementarity definition to n services, many cases of the compatibility check are possible depending on the number of services set as complementers and those set as complemented services. In the most general case, the definition of the *UC* compatibility notion, for n services, requires to check whether there are m services (complementers) which complement $n - m$ other services (complemented), where $n > m > 0$. The *UC* notion is formally defined as follows:

Definition 12 (Unidirectional Complementarity Compatibility)**.** *Given a parameter strategy PS, m STSs* $STS_{i\in\{,1\ldots,m\}} = (A_i, S_i, I_i, F_i, T_i)$ *complement* $n - m$ *STSs* $STS_{j\in\{m+1\ldots n\}} = (A_j, S_j, I_j, F_j, T_j)$ *if and only if there exist* $T'_1, \ldots, T'_m$*, with* $T'_1 \subseteq T_1, \ldots, T'_m \subseteq T_m$*,* $n > m > 0$*, such that* $(A_1, S_1, I_1, F_1, T'_1), \ldots, (A_m, S_m, I_m, F_m, T'_m), (A_{m+1}, S_{m+1}, I_{m+1}, F_{m+1}, T_{m+1}),$ $\ldots, (A_n, S_n, I_n, F_n, T_n)$ *are BC compatible.*

Example 7. We illustrate the *UC* notion using protocols in Figure 8. Let us consider STS1 and STS2 as the complemented protocols and STS3 as the complementer one. The three protocols are *UC* compatible because each action in the complemented protocols has a match in the complementer one. Although there is an unmatched action c? in the state u1, the system remains compatible since STS3 is the complementer protocol, and it can hold more actions. Notice that situations like this one are very common in real-world cases in which, *e.g.*, the protocols represent a multi-client handling system where STS1, STS2, and STS3 stand for Client1, Client2, and O-Store, and

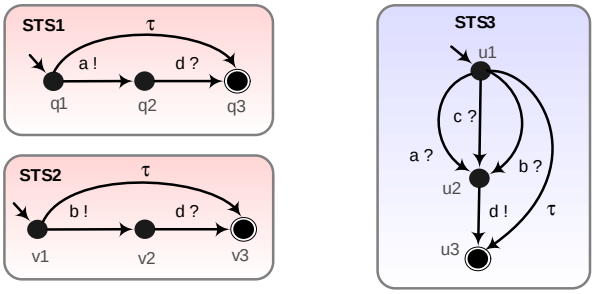


Figure 8: Unidirectional complementarity compatibility.

the messages a, b, c, and d stand for buyCD, buyEbook, buyBook, and reply, respectively.

4. Maude Encoding and Tool Support

In this section, we present, successively, a short introduction to Maude, highlighting those features of Maude relevant to our implementation, the Maude encoding of our compatibility checking framework, and the prototype tool that automates the compatibility verification.

4.1. Maude Overview

Maude [11, 12] is a high-level language and a high-performance system that supports membership equational logic and rewriting logic specification and programming of systems. Rewriting logic [33] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. Rewriting logic is parameterised by an equational logic, and therefore, Maude integrates an equational style of functional programming with rewriting logic computation. In the Maude implementation of rewriting logic, the equational logic is membership equational logic (MEL) [34, 5]. In this section, we provide an informal description of the Maude's features necessary for understanding this article, and in particular we focus on the functional features of the language, which are the ones used in the implementation of our tool.[6] The interested reader may refer to [12] for further details.

[6] Although we represent STSs as objects of a class `STS`, we do not use the Maude facilities for object-oriented programming. The implementation of the compatibility notions is given by a functional specification, and we introduce our own notation for objects and object

Membership equational logic is 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, specifications may be generic, that is, they may be defined with other specifications as parameters. This feature is key to achieve the genericity and extensibility of our framework. The requirements that a datatype must satisfy are described by *theories*. For example, sets can be constructed on top of any data, which means that its parameter could be a theory requiring only the existence of a sort. The following module `SET` defines sets over a given sort of elements (provided by the theory `TRIV`) from the constant `none` and singleton sets (by means of a subsort declaration) with an associative, commutative, and idempotent union operator `_,_`.

```
fth TRIV is
  sort Elt .
endfm
```

```
fmod SET{X :: TRIV} is
  sort Set{X} .
  subsort X@Elt < Set{X} .
  op none : -> Set{X} [ctor] .
  op _,_ : Set{X} Set{X} -> Set{X} [ctor assoc comm id: none] .
  var E : X@Elt .
  eq E, E = E .
endfm
```

The parameter `X :: TRIV` denotes that `X` is the label of the formal parameter, and that it must be instantiated with modules satisfying the requirements expressed by the theory `TRIV`. The sorts and operations of the theory are used in the body of the parameterised module, but sorts are qualified with the label of the formal parameter; thus in this case the parameter `Elt` becomes `X@Elt` in the `SET` module.

Parameterised 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. The following

configurations. The different operators implementing the notions of compatibility will take a collection of `STS` objects as an argument to check their compatibility.

view `State` maps the theory `TRIV` to the module `TRANSITION` which, among others, defines the sort `State` of STS states.

```
view State from TRIV to TRANSITION is
  sort Elt to State .
endv
```

Then, the module expression `SET{State}` denotes the instantiation of the parameterised module `SET` with the above view `State`. Note that the name of the sort `Set{X}` makes explicit the label of the parameter. In this way, when the module is instantiated with a view, like for example `State` above, the sort name is also instantiated becoming `Set{State}`.

If an MEL specification is confluent, terminating, and sort-decreasing, then it can be executed. Computation in a functional module is then 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, which can be associativity, commutativity, and identity properties. The above properties must therefore be understood in the more general context of simplification *modulo* such equational theories. Rewriting modulo allows us to deal with structures such as lists and sets very easily and efficiently, and implement many of the operations of our framework at a very high level of abstraction.

4.2. The Maude Specification

In this section, we overview the Maude specification implementing our approach. All the compatibility notions defined in the previous sections are available in our prototype framework. The only current restriction is that our implementation does not handle STSs with τ-cycles. We believe this is not an important restriction in practice, since these loops of τ hardly make sense in service descriptions because they do not correspond to any realistic implementation. The τ transitions correspond to pieces of code that a service can internally perform, *e.g.*, an internal choice, a timeout, etc. We cannot see any realistic case where a looping internal behaviour would make sense.

To improve the presentation we have omitted many details. We nevertheless give a significant account of the key pieces of the implementation, which may allow the reader to gain a sufficient understanding of our approach. As we explain below, one of the strengths of our proposal is the easiness with

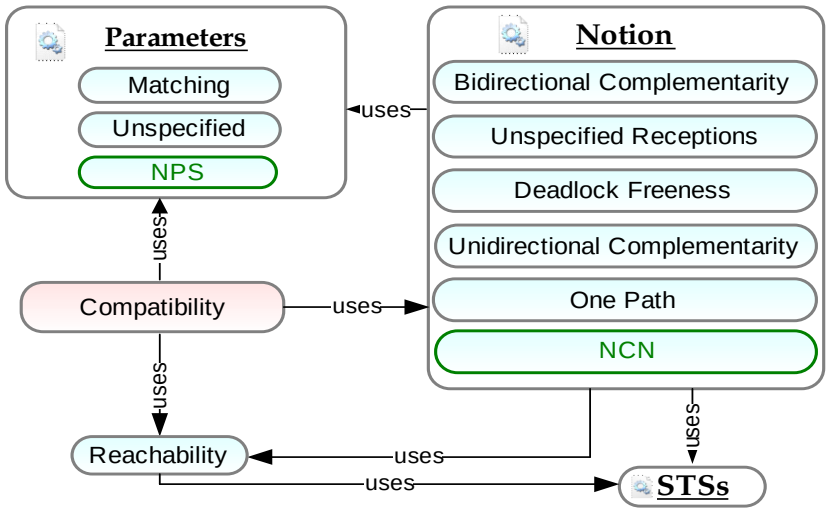


Figure 9: The framework encoding.

which new compatibility notions (NCN) and new parameter-handling strategies (NPS) can be added to the framework. Figure 9 summarises the module structure, and how the compatibility check is parameterised by the different compatibility notions and parameter-handling strategies.

STS Specification. Given sorts `Oid` of object identifiers, `Cid` of class identifiers, `Attribute` of attribute-value pairs, and `Object` of objects, and declarations

```
op STS : -> [ctor] .
op is :_ : State -> Attribute [ctor] .
op cs :_ : State -> Attribute [ctor] .
op fss :_ : Set{State} -> Attribute [ctor] .
op ts :_ : Set{Transition} -> Attribute [ctor] .

op <_:_|_> : Oid Cid Set{Attribute} -> Object [ctor] .
```

service-model STSs are represented as terms of sort `Object` —objects of a class `STS`— of the form:

```
< O : STS | is : St, cs : St', fss : StS, ts : TS >
```

An STS object has an identifier `O` (of sort `Oid`), a type (the `STS` class), and an attribute set which consists of its initial state (`is`), its current state (`cs`), a set of final states (`fss`), and a transition set (`ts`), with values of the appropriate types. Given sorts `Attribute`, `State`, and `Transition`, of object attributes, STS states, and transitions between states, respectively,

sorts `Set{Attribute}`, `Set{State}`, and `Set{Transition}` provide sets of attributes, states, and transitions, respectively, by appropriate instantiations of the sort `Set{X}`, as explained above.

A transition is represented as a tuple of the form `St1 - L -> St2` where `St1` and `St2` are, respectively, the source and target states, and `L` is its label. A label is either a τ (`tau`) or a tuple of the form `M d (PRS)`, where `M` is the message name, `d` stands for its direction, being either `?` or `!`, and `PRS` denotes the comma-separated parameter list. Each parameter in this list is represented as a term of the form `pid : pt`, where `pid` is the name of the parameter and `pt` its type.

Multisets of objects are represented as elements of the `Configuration` sort, which is defined as follows.

```
subsort Object < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
   [ctor config assoc comm id: none] .
```

Reachable States. Global states are represented as sets of STS states,[7] and sets of global states as terms of sort `Set{Set‘{State‘}}`, which is defined as the above sets, but, to avoid naming ambiguities, with `mt` as empty set and `_U_` as associative and commutative union operator with `mt` as its identity element. The $reachable_{PS}$ function (see Definition 5) is encoded as the Maude `reachable` operation, which is defined as follows:

```
var  PS : X$ParamStrat .              var  STSs : Configuration .
vars TS1 TS2 : Set{Transition} .      vars St1 St1' St1'' St2 St2' : State .
var  GStS : Set{Set‘{State‘}} .       vars O1 O2 : Oid .
vars AtS1 AtS2 : Set{Attribute} .     var  M : Message .
vars PRS1 PRS2 : List{Parameter} .

op reachable : X$ParamStrat Configuration -> Set{Set‘{State‘}} .
op reachable :
     X$ParamStrat Configuration Set{Set‘{State‘}} -> Set{Set‘{State‘}} .

eq reachable(PS, STSs) = reachable(PS, STSs, mt) .

ceq reachable(PS,
     < O1 : STS | ts : (St1 - tau -> St1', TS1), cs : St1, AtS1 > STSs,
     GStS)
  = reachable(PS,
     < O1 : STS | ts : (St1 - tau -> St1', TS1), cs : St1, AtS1 > STSs,
     reachable(PS,
       < O1 : STS | ts : (St1 - tau -> St1', TS1), cs : St1', AtS1 > STSs,
```

[7] If STS states are not all syntactically different, the sets of states of the STSs can be made disjoint by qualifying the state names with the name of the corresponding STS.

```
        (getState(STSs), St1') U GStS))
   if not (getState(STSs), St1') in GStS .
ceq reachable(PS,
       (< O1 : STS | ts : (St1 - M ! (PRS1) -> St1', TS1), cs : St1, AtS1 >
        < O2 : STS | ts : (St2 - M ? (PRS2) -> St2', TS2), cs : St2, AtS2 >
        STSs),
       GStS)
   = reachable(PS,
      (< O1 : STS | ts : (St1 - M ! (PRS1) -> St1', TS1), cs : St1, AtS1 >
       < O2 : STS | ts : (St2 - M ? (PRS2) -> St2', TS2), cs : St2, AtS2 >
       STSs),
      reachable(PS,
       (< O1 : STS | ts : (St1 - M ! (PRS1) -> St1', TS1), cs : St1', AtS1 >
        < O2 : STS | ts : (St2 - M ? (PRS2) -> St2', TS2), cs : St2', AtS2 >
        STSs),
       (getState(STSs), St1', St2') U GStS))
   if not (getState(STSs), St1', St2') in GStS
   /\ param-comp(PS, PRS1, PRS2) .
 eq reachable(PS, STSs, GStS) = GStS [owise] .
```

The `getState` operation returns the global state of a set of STSs (by collecting the current state of each of the STSs in the set). The `_in_` operation checks whether a global state belongs to a set of global states, and is here used to check whether a given state has already been visited or not. The `param-comp` operation checks the compatibility of two parameter lists according to a specified parameter-handling strategy. Note that the `reachable` operation takes arguments of sorts `X$ParamStrat` and `Configuration`. They represent, respectively, the parameter-handling strategy to be used and the set of STSs to be considered. Notice how the current states of the STSs (the `cs` attributes of the STS objects) are used to calculate the reachable states from a specific global state. Notice also that the auxiliary operation `reachable` has an additional argument that represents the set of visited states. New states reached, either with a τ transition or a synchronisation step, are added to this set. If no further states are reachable the operation terminates returning the set of reached states.[8]

State Compatibility. The *state-comp*$_{PS,d}$ function presented in Definition 7 is implemented by the `state-comp` operation, which has the following signature:

```
op state-comp : X$ParamStrat Direction Configuration -> Bool .
```

It is implemented as a predicate that takes as input a global state and checks whether it belongs to the set of compatible states (according to the specified

[8]In Maude, the equations with the `owise` attribute are attempted only if no other equation can be used.

parameter-handling strategy and direction) or not.

Compatibility Notions. The compatibility notions presented in Section 3.3 are specified by respective Maude operations. Each of these operations evaluates the compatibility of n STSs starting from their initial global state with respect to the parameter-handling strategy passed as parameter.

To illustrate the way in which they are specified, let us focus, *e.g.*, on the deadlock-freeness notion. This operation is in fact heavily used, since it is invoked from the operations implementing other compatibility notions. Given the above `reachable` operation, the `df` operator can be defined as follows:

```
op df : X$ParamStrat Configuration -> Bool .
op df :
   X$ParamStrat Configuration Set{Set`{State`}} Set{Set`{State`}} -> Bool .

var  PS : X$ParamStrat .              var  STSs : Configuration .
var  StS : Set{State} .               vars StSS StSS' : Set{Set`{State`}} .

eq df(PS, STSs) = df(PS, STSs, reachable(PS, STSs), mt) .

eq df(PS, STSs, StS U StSS, StS U StSS')
  = df(PS, STSs, StSS, StS U StSS') .
ceq df(PS, STSs, (StS U StSS), StSS')
  = false
  if not final(STSs, StS)
  /\ reachable(PS, setState(STSs, StS)) = mt .
eq df(PS, STSs, (StS U StSS), StSS')
  = df(PS, STSs, reachable(PS, setState(STSs, StS)) U StSS, StS U StSS')
  [owise] .
```

The second and third arguments of the auxiliary `df` operation represent, respectively, the set of global states to be considered, and the global states already checked. When a new state is considered, all the states reachable from it are added to the set of states in the second argument in the recursive call. The last argument allows us to avoid recomputing the checks. Thus, if a global state has already been checked, it is discarded (first equation). If a global state is a deadlock state, that is, it is not final and no state is reachable from it, then the STSs being checked are not deadlock-free and `false` is returned (second equation). In any other case (notice the `owise` attribute of the third equation), the `df` operation is recursively invoked with the global state added to the set of states already checked and the states reachable from it added to the set of states to be considered.

The other compatibility notions follow similar procedures. For instance, the *UR* notion (see Definition 9) is implemented by a function `unspecified-receptions` that takes as argument a collection of STS objects and returns a Boolean

value indicating whether the given STSs are compatible or not according to this notion.

```
op unspecified-receptions : Configuration -> Bool .
op unspecified-receptions :
     Configuration  Set{Set`{State`}} Set{Set`{State`}} -> Bool .
```

The `unspecified-receptions` operator relies on a homonymous recursive operator which has as arguments the set of STSs, the set of global states in the reachable state space not yet checked, and the set of global states already processed so that calculations are not repeated. For each new global state the function checks whether it is compatible (see the `state-comp` operator above). If it fails, the compatibility check terminates returning `false`; if it succeeds, it calls itself with the set of states reachable from the checked state added to the set of not-yet-treated reachable states and the considered state moved to the set of already-checked states.

Unified Compatibility Checking. The `COMPATIBILITY{NOTION}` module implements the compatibility definition. Its parameter `NOTION` refers to a compatibility notion encoded in the `COMPATIBILITY-NOTION{PS}` module, which is parameterised by a parameter-handling strategy `PS`.

The following *protocol-comp* function allows us to check, in a unified way, the compatibility of n given protocols described using STSs, according to a compatibility notion CN and a parameter-handling strategy PS:

$$\textit{protocol-comp} : STS_{i\in\{1,\ldots,n\}} \times CN \times PS \rightarrow \textit{Boolean}$$

If the STSs are not compatible, a counterexample CE can then optionally be computed. The framework offers two alternatives:

- *Default mode.* The counterexample consists of a sequence of global states that leads the protocols from the initial global state $(I_1, \ldots, I_n)$ to an incompatible reachable global state.
- *Verbose mode.* In addition to the sequence of global states, the compatibility check returns a set of execution traces. Each trace represents a transition sequence in one $STS_{i\in\{1,\ldots,n\}}$, leading from the initial state I_i to the state s_i in the global state $(s_1, \ldots, s_n)$ where the incompatibility is detected.

Example 8. Let us consider the results presented in Example 3. The counterexamples computed for the *normal* and *verbose* modes are as follows:

$CE_{default} = [(\mathsf{s1}, \mathsf{q1}, \mathsf{u1}), (\mathsf{s2}, \mathsf{q2}, \mathsf{u1})]$, and
$CE_{verbose}$
$= [(\mathsf{s1}, \mathsf{q1}, \mathsf{u1}), (\mathsf{s2}, \mathsf{q2}, \mathsf{u1})], \{[(\mathsf{s1}, \mathsf{a!p1{:}t1}, \mathsf{s2})], [(\mathsf{q1}, \mathsf{a?p1{:}t1}, \mathsf{q2})], [\,]\}$

These are two informative ways to understand how the incompatibility is detected. For instance, $CE_{default}$ clearly shows that the incompatibility is detected at state $(\mathsf{s2}, \mathsf{q2}, \mathsf{u1})$, which is reached from $(\mathsf{s1}, \mathsf{q1}, \mathsf{u1})$. $CE_{verbose}$ additionally returns the set of transition sequences, namely, $[(\mathsf{s1}, \mathsf{a!p1{:}t1}, \mathsf{s2})]$ in STS1, $[(\mathsf{q1}, \mathsf{a?p1{:}t1}, \mathsf{q2})]$ in STS2, and $[\,]$ in STS3, leading the protocols to $(\mathsf{s2}, \mathsf{q2}, \mathsf{u1})$.

Extensibility. In our approach, the compatibility checking framework is highly modular and extensible. New parameter-handling strategies (NPS) as well as additional compatibility notions (NCN) can be easily integrated into the framework (see Figure 9). For instance, for adding a new compatibility notion, one just needs to specify the corresponding Maude module, providing a function following our generic scheme, and add it to the list of available notions. The infrastructure in the framework will automatically consider it as one of the possibilities to be chosen.

Complexity. Given n STSs $STS_{i\in\{1\ldots n\}} = (A_i, S_i, I_i, F_i, T_i)$, $S = \prod_{i=1}^{n} |S_i|$ represents an upper bound of the number of possible global states, and $T = \sum_{i=1}^{n} |T_i|$ of the number of transitions available from any particular global state. S and T are greater than or equal to $|reachable_{PS}((I_1, \ldots, I_n))|$. The BC, UR and UC notions have a time complexity of $O(S^2 \times T^2)$, since in the worst case, for each reachable state we must check that each possible transition has a match in that global state or in any global state reachable from it (considering the rest of the protocols). The time complexity of the DF and OP compatibility notions is $O(S \times T)$. Regarding DF compatibility, we need to check that in each global reachable state the n protocols can either terminate or evolve together. In the case of OP compatibility, in the worst case, we must check all the reachable global states in order to verify if all services can at least reach one final state when interacting together.

4.3. Tool Support and Experimental Results

Our approach for checking n-service compatibility has been fully implemented in a prototype tool [38]. We present in Figure 10 an overview of the tool architecture. The Maude representations of the STSs to be considered are automatically generated using our script STS2Maude, implemented in

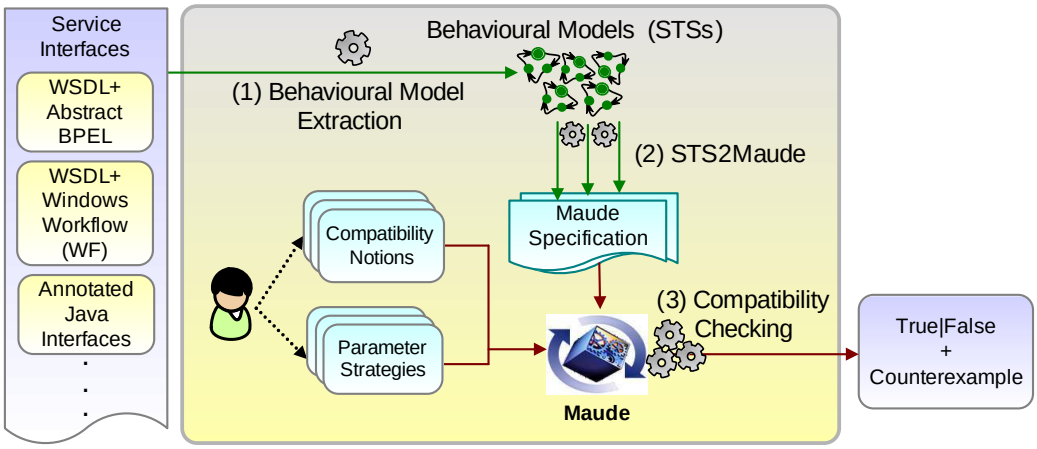


Figure 10: The compatibility checking process.

Python. Then, we execute in Maude the chosen compatibility check parameterised with a compatibility notion and parameter-handling strategy. Once the protocol compatibility is verified, a Boolean value indicates whether the n STSs are compatible according to the checked compatibility notion. If the protocols are not compatible, a counterexample is returned, in a format depending on the alternative chosen.

So far, our prototype tool has been validated on more than 180 examples, which range from small examples, to experimental boundary cases, to real-world ones. Table 2 summarises the experimental results obtained for some examples of our database. "Interfaces" shows the number of service interfaces involved in the examples. "States" and "Transitions" give the number of states and transitions, respectively, in each interface. We use "τ branchings" and "Loops" to indicate whether the protocols have τ branchings or loops ($\surd$), or not ($\times$), and to give an idea about the complexity of protocol structure. Last, "BC", "UR", "DF", and "OP" present the computation time (in seconds) needed for checking the different compatibility notions, followed by $\surd$ if the interfaces are compatible or $\times$ if not. For illustration purposes, Table 2 shows the results obtained for the symmetric compatibility notions. Experiments have been carried out on a Mac OS machine running on a 2.53 GHz Intel dual core processor with 4 GB of RAM.

As we observed during our experiments, in the case of incompatible interfaces, the checking computation time depends on the depth of the global state in which the incompatibility is detected. For instance, example Ex010

	Ex010	**Ex015**	**Ex040**	**Ex062**	**Ex097**	**Ex143**	**Ex170**	**Ex180**
Interfaces	2	2	2	4	5	10	12	13
States	86/86	86/86	172/172	4/3/2	14/3/ 4/3/3	3/5/3/4/6/ 3/7/9/4/5	3/5/86/3/4/ 86/6/3/7/9/4/6	3/2/86/6/4/86/ 8/10/4/2/5/5/4
Transitions	91/91	85/85	185/185	4/2/1	13/2/ 4/2/2	3/5/3/4/6/ 3/7/9/4/5	3/5/92/3/4/92/ 6/3/7/9/4/5	1/2/92/5/4/92/ 6/10/3/1/4/4/3
τ **branchings**	√	×	×	×	√	√	√	×
Loops	√	×	×	×	√	×	√	×
BC	14.252 | ×	1.485 | √	6.454 | √	0.006 | ×	0.074 | ×	0.280 | ×	0.797 | ×	1483.087 | ×
UR	14.216 | ×	1.261 | √	5.633 | √	0.006 | ×	0.072 | ×	0.274 | ×	0.782 | ×	1400.803 | √
DF	14.126 | ×	1.043 | √	4.790 | √	0.005 | √	0.070 | √	0.269 | ×	0.770 | ×	829.358 | √
OP	0.308 | √	0.094 | √	0.354 | √	0.001 | √	0.020 | ×	0.026 | √	0.189 | ×	6.787 | √

Table 2: Some experimental results.

is tiny compared to example Ex170. However, the number of global states reached before the incompatibility detection in Ex010 is higher than the one obtained for Ex170. As a consequence, this results in an important time difference.

In the case of compatible examples, the whole state space has to be traversed. We comment on compatibility checking time for compatible examples in the rest of this section. Running our prototype tool on small examples (protocols with less than one hundred states and transitions) requires a negligible time for checking all compatibility notions. For instance, less than 0.1 seconds was required for checking *DF* compatibility notion on examples Ex062 and Ex097. However, there is a remarkable difference in the amount of time required for bigger examples (*e.g.*, Ex040), especially those with a complex structure —protocols with hundreds of states and transitions, and many τ branchings and loops. For these examples, it can take up to several seconds to compute every compatibility result. Experiments have also shown that the time taken to analyse compatible services significantly increases with respect to the size and number of interfaces. For instance, example Ex180, which is composed of thirteen interfaces with hundreds of states and transitions, has been checked UR compatible in 1400.803 seconds. Note that our goal in this prototype was not to optimise its implementation but rather to validate and experiment with the notions formalised in Section 3, paying more attention to its genericity and extensibility than to efficiency.

The framework we propose allows us to separately check the *BC*, *UR*, *DF*, and *UC* notions. However, the computation of *BC*, *UR*, and *UC* depends on *DF*. Hence, the service incompatibility for $CN \in \{BC, UR, UC\}$ can be directly detected if the services are not *DF* compatible.

5. Related Work

There are many works dealing with the compatibility issues in the area of Service-Oriented Computing, Component-Based Software Engineering and Software architectures. However, most of the existing approaches study the compatibility for two services or components. In [47], a compatibility notion, very close to our unspecified receptions notion, is formally defined for two software components described using an automata-based formalism. More recently, [3, 10] have used a Finite State Machine (FSM) model to formalise a compatibility notion for Web services which aims at checking whether one service can substitute another. In [3], the authors adopt an asymmetric relation, namely simulation, for determining whether a new version of a service behaviour simulates a previous one. In [10] a restrictive notion of behavioural compatibility is provided saying that each trace in one Web service must also be preserved in its partner. In [29, 30, 24], the authors rely on bisimulation algorithms to define the compatibility of two Web services which are described using Petri nets. As regards process algebra, [9] and [16] have proposed a compatibility notion based on the π-calculus to ensure the successful composition of two software architectures and Web services, respectively. Bordeaux *et al.* [6] survey several compatibility notions for two services. However, the Labelled Transition System model presented in [6] does not consider value passing and internal behaviours, and they do not provide any tool support. In [4], the authors address the composability of components. They assume that two software components are composable if their respective services are pairwise compatible, where service compatibility is understood as deadlock-freeness.

The authors in [7] investigated one compatibility notion similar to that given in [47], but that can be checked for several components. Another recent work [44] defined the compatibility for two services, and then for multiple services. In [44], the compatibility analysis relies on the definition of roles (client, server) between services interacting together. Furthermore, the compatibility notion is close to our deadlock-freeness notion. However, both [7] and [44] do not consider non-observable actions and value passing. In the SOA area, [1] uses value-passing process algebra in order to verify the deadlock-freeness for a set of architectural elements. The work in [13] studies service composability based on the analysis of the execution paths of n service architectures. The composability concept, introduced in [13], consists in checking two properties referred to as *crash failure* and *no-crash*

failure. The first property is close to our deadlock-freeness compatibility notion, while the second one is similar to our one path notion.

To sum up, most of the works presented so far focus on two services, and do not consider value passing and internal behaviours in their description models. Moreover, they usually propose a unique compatibility notion useful for a specific application domain. Existing approaches often tackle the compatibility problem from a theoretical point of view, and very little attention was paid to support these contributions with automated tools [3, 10]. Our aim was to overcome the aforementioned verification limits and equip these theoretically-based approaches with some automated tool support.

6. Concluding Remarks

In this paper, we have proposed a framework for checking the compatibility of n (≥ 2) interacting service protocols taking value-passing and internal behaviours into account. Our framework goes beyond the existing approaches because it is generic, implements several compatibility notions useful for different application areas, and can easily be extended with other notions. Our proposal is fully automated inside a Maude-based prototype tool where large systems (e.g., services with hundreds of transitions and states) can be checked in a short time. In our proposal services are explored and checked at the same time, so that we stop as soon as an incompatibility is detected.

In this final section, we focus on the alternative solution mentioned in the introduction which consists in reusing existing process algebraic notations and tools. We sketch out a few ideas explaining how it would be possible to check some compatibility notions presented in this article (BC, UR, and DF) with process algebraic tools and underlying techniques, namely model-checking and equivalence-checking. For the sake of comprehension, we will give high-level explanations here, but tools such as CADP [22] or MCRL2 [23] can be used for automating these checks in practice.

- Bidirectional complementarity (BC). The idea here is to use equivalence checking techniques. Therefore, we would have to partition the set of the n involved services into two sets, m and $n-m$ (with $m < n$). Next, we would have to compose and synchronise the services in both sets, [9]

[9] By composition we mean the synchronisation of services on shared labels. Sup-

hide in the two resulting STSs these synchronisations, and finally compare one STS with the "reversal" of the other (by "reversal" we mean the STS where emissions are replaced by receptions, and receptions by emissions) with respect to an equivalence relation (probably using the observational or branching equivalence notion, but we still need to thoroughly study how to use equivalence checking techniques in the case of communicating systems, and in particular how τ transitions are supposed to be matched). Although this solution works for $n = 2$, this is not the case for $n > 2$ because additional unexpected interleavings can be obtained from the two subsystems. Another solution would be to build the composition of the n services, and check that the result contains all labels, meaning that all labels can be synchronised.

- Unspecified receptions (UR). We could first compose and synchronise all services (as described in footnote 9). In a second step, we would have to analyse the resulting STS (R) to check that all reachable emissions in each service STS appear in R. This check is quite complicated because an emission in a service STS does not need to be in R according to UR if this emission is never reached in the global system. Such a check could be achieved using an *ad-hoc* algorithm traversing each STS and R at the same time, but we do not see how process algebraic tools could be used to compute this check.

- Deadlock freeness (DF). This check can be automated by composing all the involved services, and calling the deadlock search available in existing tools. Note that DF must also be checked for the two aforementioned notions.

Now, we would like to make a short comparison between our current solution and this alternative approach using process algebraic tools. First of all, there are parts of the compatibility checks for which we do not see how process algebraic tools would help (see the comments above on the verification of the UR compatibility notion for example). Moreover, tools such as CADP or MCRL2 rely on enumerative approaches (all values that can

pose we have four services A, B, C, D and a parallel composition operator $S_1||_{SS}S_2$ meaning that S_1 and S_2 synchronise on labels belonging to the synchronisation set SS. We compute the composition of the four services A, B, C, D as follows: $A||_{\Sigma_A \cap (\Sigma_B \cup \Sigma_C \cup \Sigma_D)}(B||_{\Sigma_B \cap (\Sigma_C \cup \Sigma_D)}(C||_{\Sigma_C \cap \Sigma_D}D))$.

be exchanged between services are generated in the resulting transition system), but this is not what we want here since we need to keep a symbolic treatment of parameters in order to check that their types match. On the other hand, Maude's expressiveness and facilities for formal reasoning and searching have allowed us to implement the different notions and strategies at a very high level of abstraction, thus leading to a flexible and extensible general framework. Moreover, the implementation of our prototype tool is almost-zero distance from the formal definitions, and has allowed us to automatically experiment and test the different compatibility notions presented here.

Although protocol traversal and reachability analysis are well-understood techniques for compatibility checking, their automation can be very costly in practice. Using Maude's rewriting engine, service protocols can be incrementally traversed and checked in a simple and intuitive process. This on-the-fly-based checking is more efficient and useful than the exhaustive (global) checking which, in contrast, requires the construction of all possible execution traces of STSs before checking their compatibility. In addition, in the case of process algebraic tools, we may need to traverse the involved services several times to build STSs resulting from a composition, hiding unnecessary actions, renaming actions, checking temporal properties or equivalences, and so on. Thus, the incremental checking implies less time and space complexity, allowing us to detect incompatibility issues even for large systems.

Last but not least, our Maude implementation is modular and makes its extension with other compatibility notions possible, whereas we have not seen yet how to achieve such genericity with process algebraic tools since the verification of each compatibility notion does not rely on a common architecture as specified with Maude. To conclude, we think that Maude is a better option than process algebraic tools in terms of feasibility (in particular in the case of n services, $n > 2$), performance, and genericity.

As far as future work is concerned, we first plan to propose a high-level language which allows users to define their own compatibility notions, and some encoding techniques that would automatically generate the corresponding Maude code needed to verify such notions. Another perspective comes from the fact that the Boolean compatibility result does not show all the compatibility issues that may occur between service protocols. Therefore, we are currently working on some techniques to measure the compatibility degree of service protocols. Compatibility measuring goes further than

Boolean compatibility by detecting all existing mismatches, and computing the compatibility degree of two (or more) protocols.

Acknowledgements. We thank Radu Mateescu, Ernesto Pimentel and Miguel Palomino for their fruitful discussions and comments on this topic. We would also like to thank the anonymous referees for their very insightful comments and suggestions.

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) and FEDER, and by the regional government of Andalucía through project P07-TIC-03184.

References

[1] A. Aldini and M. Bernardo. On the Usability of Process Algebra: An Architectural View. *Theoretical Computer Science* 335(2-3):281–329, 2005.

[2] A. Arnold. Finite Transition Systems: Semantics of Communicating Systems. *Prentice Hall*, 1994.

[3] A. Aït-Bachir, M. Dumas, and M.C. Fauvet. BESERIAL: Behavioural Service Interface Analyser. In M. Dumas, M. Reichert, M.-C. Shan, eds., *Proc. of BPM'08*, vol. 5240 of LNCS, pp. 374–377. Springer, 2008.

[4] C. Attiogbé, P. André, and G. Ardourel. Checking Component Composability. In W. Löwe and M. Südholt, eds., *Proc. of SC'06*, vol. 4089 of *LNCS*, pp. 8–33. Springer, 2006.

[5] A. Bouhoula, J. P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science* 236(1):35–132, 2000.

[6] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In M.-C. Shan, U. Dayal, and M. Hsu, eds., *Proc. of TES'04*, vol. 3324 of LNCS, pp. 15–28. Springer, 2004.

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

[8] J. Cámara, J. A. Martin, G. Salaün, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel. ITACA: An Integrated Toolbox for the Automatic Composition and Adaptation of Web Services. In *Proc. of ICSE'09*, pp. 627-630. IEEE, 2009.

[9] C. Canal, E. Pimentel, and J. M. Troya. Compatibility and Inheritance in Software Architectures. *Science of Computer Programming* 41(2):105–138, 2001.

[10] 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.

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

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. vol. 4350 of LNCS. Springer, 2007.

[13] V. Cortellessa and P. Potena. Path-Based Error Propagation Analysis in Composition of Software Services. In M. Lumpe and W. Vanderperren, eds., *Proc. of SC'07*, vol. 4829 of *LNCS*, pp. 97–112. Springer, 2007.

[14] 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 M. Lumpe and E. Madelaine, eds., *Proc. of FACS'07*, *ENTCS* 215:39–55. Elsevier 2008.

[15] L. de Alfaro and T. Henzinger. Interface Automata. In A. Min Tjoa and V. Gruhn, eds., *Proc. of ESEC/FSE'01*, pp. 109–120. ACM Press, 2001.

[16] S. G. Deng, Z. Wu, M. Zhou, Y. Li, and J. Wu. Modeling Service Compatibility with Pi-Calculus for Choreography. In D. W. Embley, A. Olivé, and S. Ram, eds., *Proc. of ER'06*, vol. 4215 of *LNCS*, pp. 26–39. Springer, 2006.

[17] F. Durán, M. Ouederni, and G. Salaün. Checking Protocol Compatibility using Maude. In G. Salaün and M. Sirjani, eds., *Proc. of FOCLASA'09*, vol. 255 of *ENTCS*, pp. 65–81. 2009.

[18] L. Frantzen, J. Tretmans, and T. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, eds., *Proc. of FATES/RV'06*, vol. 4262 of *LNCS*, pp. 40–54. Springer, 2006.

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

[20] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In S.I.. Feldman, M. Uretsky, M. Najork, and C.E. Wills, eds., *Proc. of WWW'04*, pp. 621–630. ACM Press, 2004.

[21] X. Fu, T. Bultan, and J. Su. Synchronizability of Conversations among Web Services. *IEEE Transactions in Software Engineering* 31(12):1042–1055, 2005.

[22] H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In W. Damm and H. Hermanns, eds., *Proc. of CAV'07*, vol. 4590 of LNCS, pp. 158–163. Springer, 2007.

[23] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. The Formal Specification Language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, eds., Methods for Modelling Software Systems (MMOSS), Dagstuhl Seminar Proceedings, 2007.

[24] N. Hameurlain. Flexible Behavioural Compatibility and Substitutability for Component Protocols: A Formal Specification. In M. Hinchay and T. Margarita, eds., *Proc. of SEFM'07*, pp. 391–400. IEEE Computer Society, 2007.

[25] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science* 138(2):353–389, 1995.

[26] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions in Software Engineering* 23(5):279–295, 1997.

[27] A. Ingolfsdottir and H. Lin. A Symbolic Approach to Value-passing Processes. *Handbook of Process Algebra*, Chapter 7, pp. 427–478. Elsevier, 2001.

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

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

[30] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing Compatibility of BPEL Processes. In *Proc. of AICT/ICIW'06*, pp. 147–156. IEEE Computer Society, 2006.

[31] J. A. Martín and E. Pimentel. Automatic Generation of Adaptation Contracts. In *Proc. of FBTC'08*, *ENTCS* 229(2):115–131. Elsevier, 2009.

[32] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In A. Bouguettaya, I. Krueger, and T. Margaria, eds., *Proc. of ICSOC'08*, vol. 5364 of LNCS, pp. 84–99. Springer, 2008.

[33] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96(1):73–155, 1992.

[34] J. Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In F. Parisi-Presicce, ed. *Recent Trends in Algebraic Development Techniques*, vol. 1376 of *LNCS*, pp. 18–61. Springer, 1998.

[35] R. Milner. A Calculus of Communicating Systems. Springer, 1980.

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

[37] S. Nain and M. Y. Vardi. Branching vs. Linear Time: Semantical Perspective. In K. S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, eds., *Proc. of ATVA'07*, vol. 4762 of LNCS, pp. 19–34. Springer, 2007.

[38] M. Ouederni. Maude Compatibility Checker. Available at `http://www.lcc.uma.es/~meriem/tools.html`.

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

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

[41] P. Poizat and J. C. Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. *Journal of UCS* 12(12): 1741-1782, 2006.

[42] P. J. G. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. In *Proc. of the IEEE*, vol. 77, num. 1, pp. 81–98. IEEE, 1989.

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

[44] Y. Shi, L. Zhang, F. Liu, L. Lin, and B. Shi. Compatibility Analysis of Web Services. In A. Skowron, R. Agrawal, M. Luck, T. Yamaguchi, P. Morizet-Mahoudeaux, J. Liu, N. Zhong, eds., *Proc. of WI'05*, pp. 483-486. IEEE Computer Society, 2005.

[45] K. Scribner. Microsoft Windows Workflow Foundation Step by Step. *Microsoft Press*, 2007.

[46] O. Sokolsky, S. Kannan, and I. Lee. Simulation-Based Graph Similarity. In H. Hermanns and J. Palsberg, eds., *Proc. of TACAS'06*, vol. 3920 of LNCS, pp. 426–440. Springer, 2006.

[47] D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems* 19(2):292–333, 1997.

[48] P. Y. H. Wong and J. Gibbons. Verifying Business Process Compatibility. In H. Zhu, ed., *Proc. of QSIC'08*, pp. 126–131. IEEE Computer Society, 2008.

[49] W. M. Wonham and P. J. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization* 25(3):pp. 637–659, 1987.