HAL
open science

Tau Be or not Tau Be? - A Perspective on Service Compatibility and Substitutability

Meriem Ouederni, Gwen Salaün

To cite this version:

Meriem Ouederni, Gwen Salaün. Tau Be or not Tau Be? - A Perspective on Service Compatibility and Substitutability. International Workshop on Component and Service Interoperability (WCSI'10), Jun 2010, Malaga, Spain. pp.57–70. inria-00539099

HAL Id: inria-00539099

https://inria.hal.science/inria-00539099v1

Submitted on 24 Nov 2010

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

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

Tau Be or not Tau Be?
A Perspective on Service Compatibility and Substitutability

Meriem Ouederni
University of Málaga, Spain
meriem@lcc.uma.es

Gwen Salaün
Grenoble INP–INRIA–LIG, France
Gwen.Salaun@inria.fr

One of the main open research issues in Service Oriented Computing is to propose automated techniques to analyse service interfaces. A first problem, called compatibility, aims at determining whether a set of services (two in this paper) can be composed together and interact with each other as expected. Another related problem is to check the substitutability of one service with another. These problems are especially difficult when behavioural descriptions (*i.e.*, message calls and their ordering) are taken into account in service interfaces. Interfaces should capture as faithfully as possible the service behaviour to make their automated analysis possible while not exhibiting implementation details. In this position paper, we choose Labelled Transition Systems to specify the behavioural part of service interfaces. In particular, we show that internal behaviours (τ transitions) are necessary in these transition systems in order to detect subtle errors that may occur when composing a set of services together. We also show that τ transitions should be handled differently in the compatibility and substitutability problem: the former problem requires to check if the compatibility is preserved every time a τ transition is traversed in one interface, whereas the latter requires a precise analysis of τ branchings in order to make the substitution preserve the properties (*e.g.*, a compatibility notion) which were ensured before replacement.

1 Introduction

The definition of Interface Description Languages (or contract languages) which provide a good trade-off between expressiveness and abstraction level is not a recent research topic. With the advent of Component-Based Software Engineering in the 90s, many works were dedicated to the design of such languages, see for instance [4, 1, 11, 29]. This work took a new breath with the recent venue of (Web) services. Indeed, although the black-box nature of components can be source of discussion [9, 25, 30], this is not the case of services since they are deployed and available on-line, therefore their internal implementation has no reason to be accessible to users.

Existing IDLs distinguish four interoperability levels [2]: signature, behaviour (or interaction protocol), semantics, and quality of service. In this paper, we focus on the behavioural description level. This level has often been emphasised as crucial [29, 12] because an *a-priori* knowledge of every service control flow is essential to avoid *a-posteriori* erroneous executions (such as deadlock) of a set of interacting services. In the services area, several notations (Petri nets, transition systems, process algebras, state diagrams, etc.) have already been proposed to specify the behavioural part of service interfaces. Here, we have chosen Labelled Transition Systems which is a simple yet expressive model often used as semantic foundation to higher-level formalisms. In this model, we also consider internal or non-observable behaviours using τ transitions. Internal behaviours are important because analysing service interfaces may show that they will interact correctly if observable behaviours only are considered whereas they will actually behave erroneously due to internal behaviours. These τ transitions correspond to abstractions of pieces of code (*e.g.*, conditions involving variables and functions). This information may be preserved

Cámara, Canal, and Salaün (Eds.)
Component and Service Interoperability (WCSI10)
EPTCS 37, 2010, pp. 57–70, doi:10.4204/EPTCS.37.5

© M. Ouederni & G. Salaün
This work is licensed under the
Creative Commons Attribution License.

and may help the designer who wants to compose a set of services together. However, as far as automatic approaches are concerned, the analysis of such conditions is difficult because it deserves to have a full understanding of types and functions used in those guards.

Once the interface model is defined, several issues have to be worked out and are still actively studied in the service research community: service discovery, automatic composition, validation and verification, adaptation, etc. Here, we focus on two problems (related to one another) referred to as service compatibility and substitutability (or replaceability). The first problem aims at checking whether two (or more) services are *compatible*, that is can interact *properly* until reaching a correct termination state. Several notions of behavioural compatibility have already been proposed in the literature. In this paper, we will use three notions for illustration purposes, namely deadlock-freeness [11], unidirectional-complementarity and unspecified-receptions [34, 7]. The second problem aims at checking if one service can be *substituted* with another while ensuring that the reconfigured system will behave *similarly* from a behavioural point of view.

Our goal in this paper is to focus on τ transitions and first show that such transitions are necessary in interface models to avoid erroneous behaviours. Second, we will show that when checking compatibility and substitutability, these τ transitions have to be handled *correctly*. One of the main objectives of this paper is to explain what is meant by *correctly* (or *properly*, *similarly*, which are adverbs used before in this introduction). In particular, their analysis is different from one problem to another. Compatibility requires to check that observable actions satisfy the compatibility notion every time an internal behaviour is traversed in one of the two involved services. On the other hand, the substitutability verification requires a precise analysis of τ branchings in order to check that the new service behaves as its former version. We will illustrate our arguments throughout this paper with some simple examples.

Our objective is not to present some algorithms and tools to automate those checks. Such algorithms can be found in related papers, *e.g.*, [11, 1, 13]. As far as tool support is concerned, the compatibility check can be implemented using Maude [15] as presented in [17], and the substitutability check can be achieved using CADP [19] and the Bisimulator tool [3].

The rest of the paper is organized as follows. Section 2 introduces our behavioural model of services. Section 3 first presents some compatibility notions we use in this paper for illustration purposes, and then discuss the way τ transitions should be handled. In Section 4, we tackle the substitutability problem, and present some solutions to check the substitutability of services with a special focus on τ transitions. Finally, we overview existing works in Section 5 and draw up some conclusions in Section 6.

2 Model of Service Interfaces

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) which is a Labelled Transition System (LTS) extended with value passing (data parameters coming with messages). More formally, an STS is a tuple (A,S,I,F,T) where: A is an alphabet which corresponds to the set of labels associated to transitions, S is a set of states, $I \in S$ is the initial state, $F \subseteq S$ is a nonempty set of final states, and $T \subseteq S \backslash F \times A \times S$ is the transition relation. In our model, a *label* is either a τ (internal action) or a tuple (m,d,pl) where m is the message name, d stands for the communication direction (either an emission ! or a reception ?), and pl is either a list of data terms if the label corresponds to an emission, or a list of variables if the label is a reception.

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 transitions (in-

ternal choice). Another possibility would be to keep choice conditions as part of the model (as done in Symbolic Transition Graph introduced in [24]), and analyse them using subtyping relations, see [14] for instance. However, in the general case, it is not possible to analyse boolean expressions used in guards because they can involve variables and functions, and at design-time, we do not know variable values. Therefore, there is no way to predict how a choice will behave at run-time. This is why choice or loop conditions are often made abstract and specified as τ transitions in behavioural interfaces.

In our model, no transition can go out from a final state because, in (Web) services, an implementation explicitly defines a termination construct (*e.g.*, Terminate in BPEL), and therefore the corresponding transition system consists of a transition labelled with τ followed by a final state. Such τ transition can be minimized if it appears in a sequence (τ-confluence), but this is not the case if it is involved in a branching structure (a state with several outgoing transitions).

Synchronizations between services respect a synchronous and binary communication model. Therefore, two services synchronize if one can evolve through an emission, the other through a reception, and both labels have the same message and matching parameters (same number of parameters with the same type and in the same order). Internal behaviour cannot be controlled because this corresponds to an independent evolution of a service, *i.e.*, a service can internally decide to change its state without any apparent or observable reason. The operational semantics of STS is formalised in [17].

This model is simple yet offers a good abstraction level for describing and analysing service behaviours. Moreover, STSs can be easily derived from higher-level description languages such as Abstract BPEL, see for instance [18, 32, 10] where such abstractions were used for verification, composition or adaptation of Web services. In the rest of the paper, we will describe service interfaces only with their corresponding STSs. Signatures can be deduced from the argument types appearing in STS labels.

Internal Behaviours. Service analysis could be worked out without taking into account their internal evolution because that information is not observable from its partners point of view (black-box assumption). However, keeping an abstract description of the non-observable behaviours while analysing services helps to find out possible interoperability issues. Indeed, although one service can behave as expected by its partner from an external point of view, interoperability issues may occur because of unexpected internal behaviours that services can execute. For instance, Figure 1 shows two versions of one service protocol without (S1) and with (S1') its internal behaviour. As we can see, S1 and S2 can interoperate on a and terminate in final states (b! in S1 has no counterpart in S2 and cannot be executed, this is due to synchronous communication). However, if we consider S1', which is an abstraction closer to what the service actually does, we see that this protocol can (choose to) execute a τ transition at state s1 and arrives at state s3 while S2 is still in state u1. At this point, both S1' and S2 cannot exchange messages, and the system deadlocks. This issue would not have been detected with S1.

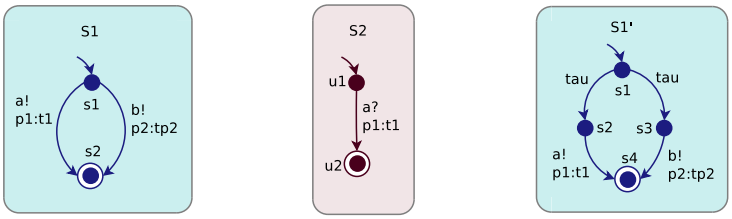


Figure 1: S1 and S2 interoperate successfully, but S1' and S2 can deadlock

Now, let us focus on higher-level languages, such as abstract BPEL or abstract Windows workflow (WF), which are used in the literature [28, 16, 27] as abstract descriptions (Interface Description Lan-

guages) of service behaviours. Here we choose WF to illustrate how STSs and in particular τ transitions are extracted from this workflow-based notation. WF describes service behaviours using a set of basic activities, *e.g.*, IfElse, Listen and While, for which it is useful to keep some τ transitions in their respective STS descriptions.

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

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

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

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

3 Compatibility

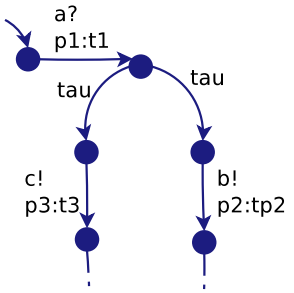
In this section, we first present three notions of compatibility, namely deadlock-freeness, unidirectional-complementarity and unspecified-receptions. We have chosen these notions because they are simple to understand, and often used by related work in the literature [7, 34, 11, 6, 17]. We will use them in the rest of this paper to illustrate the discussion. In the second half of this section, we point out the subtleties of dealing with τ transitions when checking behavioural compatibility.

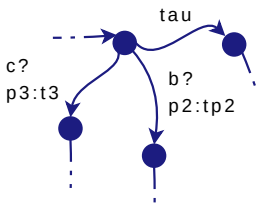
3.1 Compatibility Notions

Deadlock-freeness. This notion says that two service protocols are compatible if and only if, starting from their initial states, they can evolve together until reaching final states. Figure 2 presents a simple example to illustrate this notion. S1 and S2 are not compatible because after interacting on action a, both services are stuck. On the other hand, S1' and S2 are deadlock-free compatible since they can interact successively on a and c, and then both terminate into a final state.

Unidirectional-complementarity. Two services are compatible with respect to this notion if and only if there is one service which is able to receive (send, respectively) all messages that its partner expects to send (receive, respectively) at all reachable states. Hence, the "bigger" service may send and receive more messages than the "smaller" one. Additionally, both services must be free of deadlocks. This notion is different to what is usually called simulation or preorder relation [31] because the two protocols under analysis here aim at being composed, and accordingly present opposite directions. However, both definitions share the inclusion concept: one of the two protocols is supposed to accept all the actions that the other can do. Figure 3 first shows two services S1 and S2 which respect this unidirectional-complementarity compatibility: all actions possible in S1 can be captured by S2. However, S2 does not complement S1' because S2 is not able to synchronize on action c with S1'.

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,...)`
`)`
	

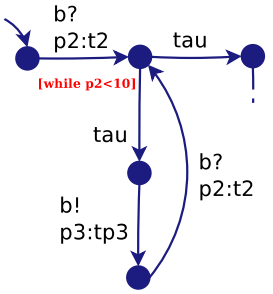
`WebServiceInput(b?(p2:t2))`
`While`
`(`
`(p2 < 10),`
`InvokeWebService`
`(b!(p3:t3), b?(p2:t2))`
`)`
`...`
	

Table 1: Examples of abstract WF activities and their corresponding STSs

Unspecified-receptions. This definition requires that if one service can send a message at a reachable state, then the other service must receive that emission. Furthermore, one service is able to receive messages that cannot be sent by the other service, *i.e.*, there might be additional unmatched receptions. 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 when protocols interact together. Additionally, both services must be free of deadlocks. In Figure 4, S1 and S2 are not compatible because S1 cannot receive all actions that S2 can send (c!). But S1' and S2 are compatible because all emissions on both sides have a matching reception on the other.

The reader interested in the formal definitions for these compatibility notions can refer to [6, 17].

3.2 Internal Behaviours

Compatibility checking verifies that two interacting services fulfill each other's requirements. Interaction between services basically depends on synchronisations over observable actions and then can be defined

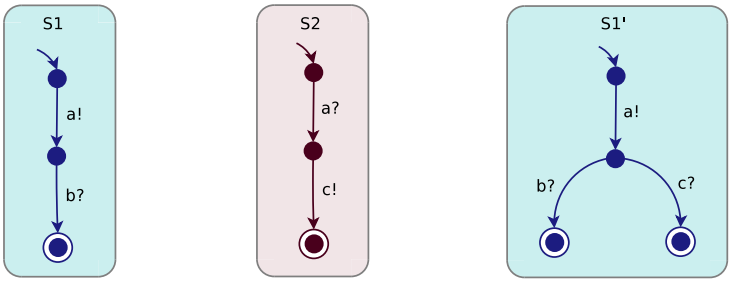


Figure 2: Deadlock-freeness compatibility

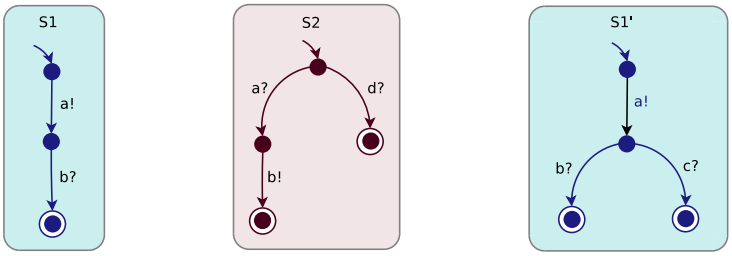


Figure 3: Unidirectional-complementarity compatibility

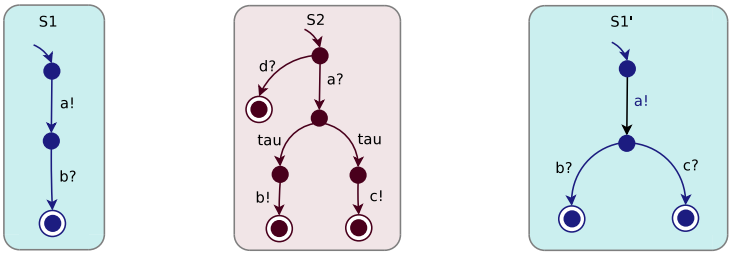


Figure 4: Unspecified-receptions compatibility

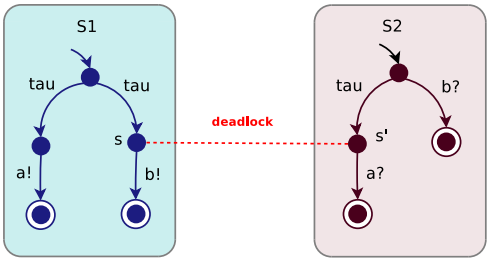


Figure 5: τ transitions need to be analysed

using a criterion set on these observable actions (see for instance the compatibility notions we presented previously). Since services can evolve independently through some non-controllable τ transitions, the behavioural compatibility requires that each internal evolution must lead both services into a state in which the criterion is satisfied. This means that every time a τ transition is traversed in one of the two STSs, the compatibility must be checked again on the target state. This way to process τ transitions leads to a unique way to handle them all along the compatibility checking. This is not the case in the context of service substitution where services can be compared according to different ways of dealing with their internal behaviours, similarly to what is achieved in equivalence checking [31, 8, 21] (see Section 4 for more details on the substitutability problem).

Let us illustrate these ideas on a couple of examples. First of all, Figure 5 shows that it is not enough to focus on observable actions when checking service compatibility: τ transitions must be analysed as well. In this example, both services can interact on a and b from an observational point of view, *i.e.*, considering only observable traces without τ transitions. However, if the compatibility check does not analyse only observational actions but also internal ones, a deadlock is detected when services S1 and S2 move to state s and s', respectively, by executing a τ transition. Therefore, these two services are not deadlock-free compatible.

The question now is: how τ transitions are supposed to be analysed when checking compatibility? Similarly to equivalence checking, one may want to match τ transitions appearing in both service interfaces together. As an example, observational (or weak) equivalence [31] checks that one τ on one side matches with a sequence of zero or more τ on the other. Figure 6 shows an example in which the state matching respects this weak relation[1]. Nevertheless, these services are not compatible with respect to the unspecified-receptions compatibility[2], because S1 can evolve to state s by executing a τ transition and S2 to s', and in this configuration, action b! in S1 has no counterpart in service S2 in state s'.

To sum up, in order to check compatibility, τ transitions need to be analysed, and one has to check after each τ transition that the compatibility notion is verified by the forthcoming observational actions. We also claim that reasoning on the τ branchings as done in equivalence checking (matching τ appearing in both interfaces) is not useful when one checks compatibility. Indeed, in order to ensure correct interactions, we do not want to match one service internal actions with those of its partner. This would be meaningless in a composition situation because these actions are non-controllable from a partner point of view, and do not have anything to see with one another. We only need to check that their observable

[1]Actually, services S1 and S2 are equivalent *wrt.* the observational relation if directions in one service are reversed as follows: $\overline{l!} = l?$, $\overline{l?} = l!$. In our model, messages may come with parameters, and this check would also require to remove parameters beforehand.

[2]Here, we chose a special example where no additional receptions appear in both services, and the parallel with equivalence checking is therefore easier to make.

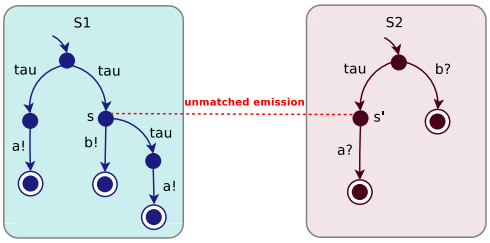


Figure 6: τ transitions matching is not necessary

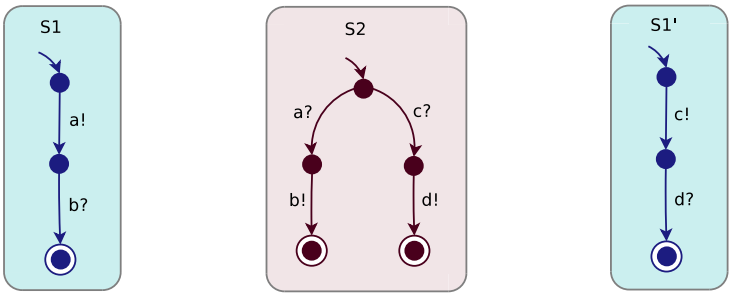


Figure 7: Checking substitutability as compatibility may be misleading

actions behave as defined in the compatibility notion in spite of possible τ transitions.

4 Substitutability

Substitutability aims at replacing one service embedded as part of a larger system with another service such that the entire system is able to interact as before (*i.e.*, respecting the same compatibility notion). In this paper, we focus on context-dependent approaches, *i.e.*, where partners (sometimes called environment) are defined and known.

First of all, the substitutability problem has two formulations in the literature. Suppose we have a system consisting of two services S1 and S2, and both services are compatible *wrt.* a compatibility notion C. Imagine now that we want to substitute service S1 with a new service S1'. A first way to check if this substitution is possible is to verify that S1' and S2 are compatible *wrt.* compatibility C. A second way is to compare S1 and S1' to ensure they are related by a certain *relation*. Both solutions are valid, however the first formulation can be misleading and for this reason, we will focus on the second in the rest of this section. To illustrate this point, see the example given in Figure 7 where we consider for example the unspecified-receptions compatibility. S1 and S2 are compatible *wrt.* this notion. As far as the first formulation above is considered, S1' can substitute S1 because S1' is compatible with S2 (no reachable emissions without counterpart in both protocols). Nevertheless, S1 and S1' have completely different behaviours and therefore fulfill different objectives as well (imagine for instance that action a corresponds to a search in a database, and action c to a modification of that database).

In our model, we chose a level of abstraction where we replace guards with τ transitions. An alternative approach is to keep guards and use subtyping techniques, see for instance [14], for analysis purposes when checking the substitutability problem. Since our model considers τ transitions, we can use strong notions such as equivalences [31] (or bisimulations), or more flexible ones such as simulation [31] or

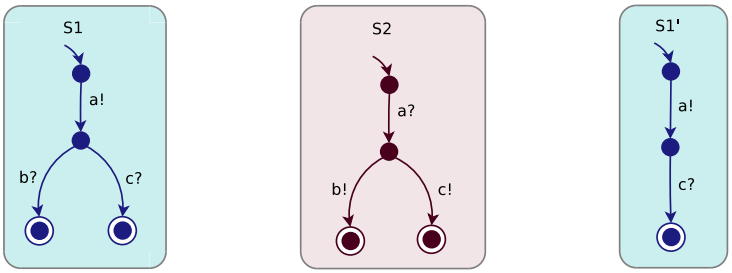


Figure 8: Simulation is too weak to check substitutability

behavioural subtyping [26][3]. In the general case (for any compatibility definition as one of those presented in Section 3), simulation or subtyping can be too loose. Let us focus first on the simulation (or preorder) notion. Intuitively, all the actions possible in one transition system must appear in the other. In such a case, we say that the "bigger" protocol simulates the "smaller". In Figure 8, we show that this notion is too weak to always preserve compatibility. This example shows two services S1 and S2 which are unspecified-receptions compatible. However, if we replace S1 with S1' where S1 simulates S1', then there is an emission (b!) in S2 which has no counterpart in S1', therefore S1' and S2 are not compatible.

Regarding behavioural subtyping, different definitions exist in the literature for the substitutability problem, see for instance [11, 5, 23]. We illustrate here with the definition proposed in [5] where *"the algorithm for substitutivity checking verifies that service A demands fewer and fulfills more constraints than service B"*. In terms of transition systems, this means that a service can replace another if it can have more receptions and less emissions. Again, in the general case, this definition is not strong enough. Figure 9 gives a simple example where two services S1 and S2 are deadlock-free compatible, but S1' and S2 are not, even if S1' is a behavioural subtype of S1 according to the definition quoted above. Let us emphasise here that our claim focuses on the general case (any compatibility notion). If we consider a precise compatibility notion, it can be demonstrated that this behavioural subtyping relation is enough. This is the case for instance with the unspecified-receptions compatibility (see Figure 10 for an example) because the new service can have more receptions and less emissions. As a consequence, all the emissions in the service which does not change are still captured (the new service preserves all its former receptions and may have more). Moreover, the new service can only have less emissions compared to its former version, and since all the emissions in the old service had a counterpart in its partner, the new service will have corresponding receptions as well.

Equivalences are strong yet suitable relations to check the substitutability problem, because they preserve all observable actions and then the compatibility notion should be preserved as well. However, different equivalence relations exist, and they handle differently internal behaviours. In this paper, we will focus on some well-known equivalence relations, namely strong, branching, weak, and trace equivalence, from the strongest to the weakest notions (see [20] for more details on these notions and their formal relationship).

As far as substitutability is concerned, a strong equivalence or bisimulation [31] is too strong because it requires to match not only observable actions but also τ transitions. Perfectly matching these internal transitions does not make sense in the Web services area because two service implementations can sligthly differ yet exhibit exactly the same behaviour from an external point of view.

[3]Refinement is also a notion used for the substitution problem, see [5, 23] for example. This notion is stronger than subtyping [23], but we will not talk about it in this paper because our goal is to focus on τ handling and not to give a survey on substitution notions.

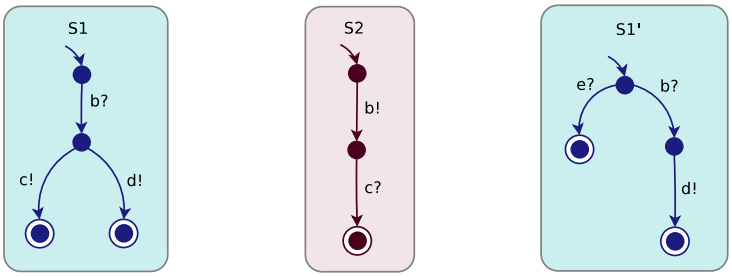


Figure 9: Behavioural subtyping is too weak in the general case

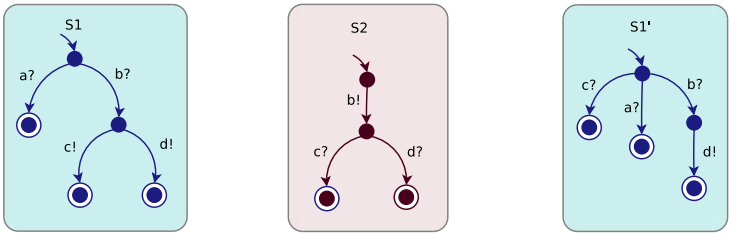


Figure 10: Behavioural subtyping works for precise compatibility notions, *e.g.*, unspecified-receptions

At the other extremity, trace equivalence is too weak because this relation only analyses the observable behaviour, and does not preserve compatibility. Figure 11 shows an example where S1 and S2 are two services respecting a deadlock-freeness compatibility. However, if we replace S1 by S1', even if S1 and S1' are trace equivalent, S1' and S2 are not deadlock-free compatible because a deadlock occurs if S1' decides to execute the τ transition.

Weak and branching equivalences are the strongest of the weak equivalences [20]. These two notions preserve behavioural properties (does not add deadlocks for instance) on observable actions. Consequently, these two equivalence relations are adequate to verify the substitutability of one service by another. Such results were formally proven for weak equivalence [13]. Branching equivalence is stronger than weak equivalence, and is checked more efficiently from a computational point of view, so it should be preferred if huge protocols are involved when checking substitutability.

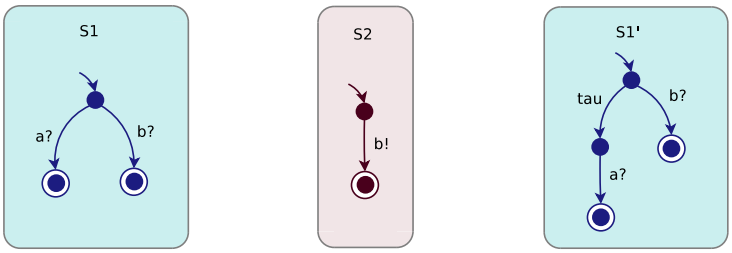


Figure 11: Trace equivalence is too weak to check substitutability, and does not preserve compatibility

5 Related Work

In this section, we overview works existing on the compatibility and substitutability questions, with a specific focus on approaches handling internal behaviours in their model and solutions.

5.1 Compatibility

To the best of our knowledge, here are the approaches [11, 22, 33, 1, 17] which take τ transitions into account in their description models. In [22], the author relies on a bisimulation algorithm to define the compatibility of (Web) services which are described using Petri nets. The bisimulation-based compatibility associates τ transitions in the same way as Milner's strong equivalence. Matching τ transitions as done in the strong equivalence does not make sense when checking services compatibility in our opinion.

In [11], a compatibility notion based on the π-calculus considers two services to be compatible if they are deadlock-free. In [1], the authors rely on an automata-based model and define context-dependent compatibility. This work considers two interfaces to be compatible if their product can be composed with a third component and this composition is deadlock-free. In these two works, the authors propose to analyse τ transitions similarly to what is introduced in Section 3.2, *i.e.*, each internal evolution must lead the system into states where the deadlock-freeness is preserved.

A π-calculus description model is also used in [33] where two services are compatible if there is always at least one sequence of interactions that make them reach final states. This notion is quite weak when composing services because the deadlock-freeness property cannot be guaranteed. In [33], τ transitions only appear as the visible result of synchronisations (as defined in the π-calculus or CCS semantics). Then, two services are considered compatible if their composition can engage a sequence of τ actions until reaching final states. Therefore, no particular processing is associated to τ transitions in this approach.

In a previous paper [17], we considered an automata-based model and proposed a generic framework which automatically checks service protocols according to a compatibility notion passed as parameter. Three strategies for handling τ transitions were implemented, namely strong, weak and trace. These strategies are inspired from the ways of associating internal behaviours proposed by the strong, weak and trace equivalence relations. Considering different ways of dealing with τ transitions does not impact the result of the compatibility check, but adds an additional analysis on τ branchings.

5.2 Substitutability

Here again, our goal is not to survey all existing works but only those which use internal behaviours in their interface model. First, Hameurlain [22] addresses the substitutability of component protocols described with Petri nets. The substitutability notion used in this paper is a strong bisimulation as introduced by Milner in [31]. Replacing components using this relation enables to preserve system compatibility. However, it is a very strict relation as far as the matching of τ transitions is concerned, and weaker relations may be enough to preserve this compatibility.

In [13], the authors check component substitutability using weak bisimulation. They show that whenever there is a system in which a component is replaced with an observationally equivalent one, the system remains equivalent to the former one. This relation is less restrictive than strong bisimulation used in [22].

More recently, [14] used a Finite State Machine (FSM) model to formalise a substitutability notion for Web services which preserves compatibility. The authors consider a symmetric approach which re-

quires that services must have the same traces. In this paper, pre/post-conditions are used rather than τ transitions. Therefore, the authors compare these conditions using a subtyping relation: the pre-conditions of an old service must be simulated by those of the new service and the post-conditions of the new service must be simulated by those of the old service.

6 Concluding Remarks

In this paper, we have focused on behavioural models of service interfaces, especially those involving internal behaviours. Those behaviours are essential because if they are not taken into account in service models, the composition or substitution of services may cause erroneous executions. We have discussed various solutions to handle internal behaviours when checking compatibility and substitutability of services. Our conclusions are the following: (i) when checking compatibility, the notion to be ensured has to be verified after every internal behaviour appearing in each behavioural interface, and (ii) when checking substitutability, behavioural models need to be equivalent *wrt.* a relation stronger enough (such as weak or branching equivalence) to preserve all properties on observable behaviours.

Now, we would like to conclude with four challenges which are still some open issues in the context of the compatibility and substitutability checking: (i) generalising existing approaches to consider not only two services but a set of services (compatibility of n services, substitution of k services involved in a system by m new services, etc.), (ii) considering an asynchronous communication model (*e.g.*, based on message queues), (iii) proving that branching equivalence is better than weak equivalence when checking service substitutability, and (iv) not only returning a boolean result but, if services cannot be properly composed or replaced (*false* result), detecting the mismatches and measuring the compatibility/substitutability degree separating both protocols.

Acknowledgements. The authors thank Radu Mateescu for interesting comments on an earlier version of this paper, and Francisco Durán for fruitful discussions on this topic. This work has been partially supported by the project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science (MICINN) and FEDER.

References

[1] L. de Alfaro & T. Henzinger (2001): *Interface Automata*. In: *Proc. of ESEC/FSE'01*, ACM Press, pp. 109–120.

[2] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky & M. Tivoli (2006): *Towards an Engineering Approach to Component Adaptation*. In: *Architecting Systems with Trustworthy Components, Lecture Notes in Computer Science* 3938, Springer-Verlag, pp. 193–215.

[3] D. Bergamini, N. Descoubes, C. Joubert & R. Mateescu (2005): BISIMULATOR*: A Modular Tool for On-the-Fly Equivalence Checking*. In: *Proc. of TACAS'05, Lecture Notes in Computer Science* 3440, Springer-Verlag, pp. 581–585.

[4] A. Beugnard, J.-M. Jézéquel & N. Plouzeau (1999): *Making Components Contract Aware*. *IEEE Computer* 32(7), pp. 38–45.

[5] D. Beyer, A. Chakrabarti & T. A. Henzinger (2005): *Web Service Interfaces*. In: *Proc. of WWW'05*, pp. 148–159.

[6] L. Bordeaux, G. Salaün, D. Berardi & M. Mecella (2004): *When are Two Web Services Compatible?* In: *Proc. of TES'04, Lecture Notes in Computer Science* 3324, Springer-Verlag, pp. 15–28.

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

[8] S. D. Brookes, C. A. R. Hoare & A. W. Roscoe (1984): *A Theory of Communicating Sequential Processes*. *Journal of the ACM* 31(3), pp. 560–599.

[9] M. Buchi & W. Weck (1999): *The Greybox Approach: When Blackbox Specifications Hide Too Much*. Technical Report 297, Turku Center for Computer Science.

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

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

[12] C. Canal, P. Poizat & G. Salaün (2008): *Model-Based Adaptation of Behavioural Mismatching Components*. *IEEE Transactions on Software Engineering* 34(4), pp. 546–563.

[13] I. Cerná, P. Vareková & B. Zimmerova (2007): *Component Substitutability via Equivalencies of Component-Interaction Automata*. In: *Proc. of FACS'07, ENTCS* 182, pp. 39–55.

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

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

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

[17] F. Duran, M. Ouederni & G. Salaün (2009): *Checking Protocol Compatibility using Maude*. In: *Proc. of FOCLASA'09, ENTCS* 255, pp. 65–81.

[18] X. Fu, T. Bultan & J. Su (2004): *Analysis of Interacting BPEL Web Services*. In: *Proc. of WWW'04*, ACM Press, pp. 621–630.

[19] H. Garavel, R. Mateescu, F. Lang & W. Serwe (2007): *CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes*. In: *Proc. of CAV'07, Lecture Notes in Computer Science* 4590, Springer-Verlag, pp. 158–163.

[20] R. J. van Glabbeek (2001): *The Linear Time - Branching Time Spectrum I*, chapter 1, pp. 3–99. Handbook of Process Algebra. Elsevier.

[21] R. J. van Glabbeek & W. P. Weijland (1996): *Branching Time and Abstraction in Bisimulation Semantics*. *J. ACM* 43(3), pp. 555–600.

[22] N. Hameurlain (2005): *On Compatibility and Behavioural Substitutability of Component Protocols*. In: *Proc. of SEFM'05*, IEEE Computer Society, pp. 394–403.

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

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

[25] J. Henriksson, F. Heidenreich, J. Johannes, S. Zschaler & U. Aßmann (2007): *How Dark Should a Component Black-box Be? The Reuseware Answer*. In: *Proc. of WCOP'07*.

[26] B. Liskov & J. M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Program. Lang. Syst.* 16(6), pp. 1811–1841.

[27] J. A. Martín & E. Pimentel (2009): *Automatic Generation of Adaptation Contracts*. In: *Proc. of FOCLASA'08, ENTCS* 229, Elsevier, pp. 115–131.

[28] R. Mateescu, P. Poizat & G. Salaün (2008): *Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques*. In: *Proc. of ICSOC'08, Lecture Notes in Computer Science* 5364, Springer, pp. 84–99.

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

[30] F. Puntigam (2007): *Black & White, Never Grey: On Interfaces, Synchronization, Pragmatics, and Responsibilities*. In: *Proc. of WCOP'07*.

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

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

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

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