Analysis and Verification of Service Interaction Protocols
– A Brief Survey –

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

Modeling and analysis of interactions among services is a crucial issue in Service-Oriented Computing. Composing Web services is a complicated task which requires techniques and tools to verify that the new system will behave correctly. In this paper, we first overview some formal models proposed in the literature to describe services. Second, we give a brief survey of verification techniques that can be used to analyse services and their interaction. Last, we focus on the realizability and conformance of choreographies.

1 Introduction

Service-Oriented Computing (SOC) has emerged as a new software development paradigm that enables implementation of Web accessible software systems that are composed of distributed services which interact with each other via the exchange of messages. In order to facilitate integration of independently developed services that may reside in different organizations, it is necessary to provide some analysis and verification techniques to check as automatically as possible that the new system will behave correctly avoiding erroneous interactions leading to deadlock states for instance.

Let us show a couple of examples to illustrate the previous arguments, where services are modelled using Labelled Transition Systems (presented more formally in Section 2). Services S1 and S2 in Figure 1 can end up into a deadlock because after interacting on a, S2 can decide to evolve through an internal action τ (right-hand branch of the choice) and is deadlocked: S1 cannot interact on c with S2 at this point. On the other hand, the execution of S1' and S2 is free of deadlocks because all emissions on both sides have a matching reception on the other. In Figure 2, suppose that S1 is a client and S2 a service. S1 is satisfied because the service is able to reply his/her request, *i.e.*, can receive a and send b. However, if we focus on another version of this client S1', after submitting a, the client expects either b or c, but S2 is not able to provide c. This is another kind of issue that one may need to detect: all the messages (in the client here) must have a counterpart.

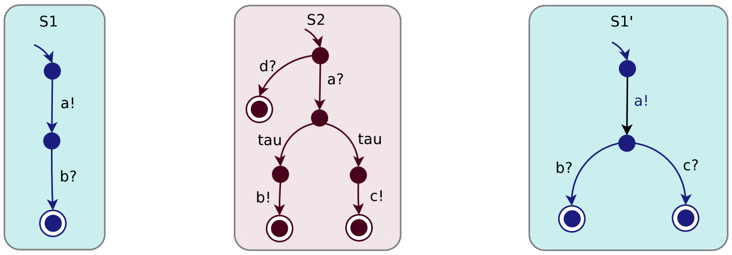

Figure 1: Deadlocking execution of services

Salaün, Fu, and Hallé (Eds.): Fourth International Workshop
on Testing, Analysis and Verification of Web Software
EPTCS 35, 2010, pp. 75–86, doi:10.4204/EPTCS.35.7

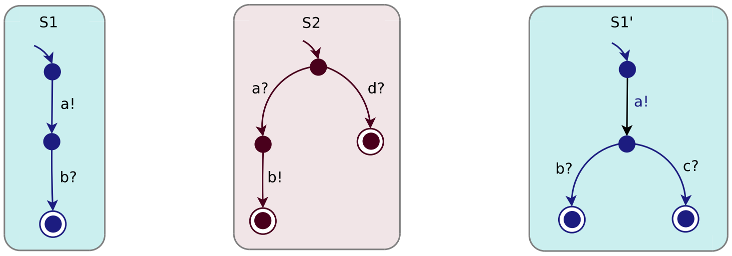

Figure 2: Unmatching messages

In this paper, we do not want to present the many works and papers which have proposed analysis and verification for Web services, this would be too long and uninteresting. Our goal is to focus on specific issues occuring in this area, and present some automated techniques to work them out. We will also give some key references for each problem to enable the reader to go deeper in these issues and solutions existing for them. Beyond giving a quick overview of service analysis techniques, we also point out at the end of the paper a few challenges that are still open, to the best of our knowledge.

The organization of this paper is as follows. First, we present in Section 2 some formal models that are often used to represent abstract descriptions of services, *e.g.*, Petri nets, automata-based models, process algebras. In Section 3, we focus on automated verification techniques, namely equivalence-checking on one hand, and temporal properties and model-checking on the other. Section 4 is dedicated to the compatibility of two (or more) services. This section also comments on some techniques to quantify the compatibility degree between two services, and on service adaptation which is a solution to work out existing mismatches detected using compatibility analysis. In Section 5, we present a slightly different kind of analysis which aims at checking the realizability (and conformance) of choreography specifications. Realizability indicates whether services can be generated from a given choreography specification in such a way that the interactions of these services exactly match the choreography specification. Finally, we draw up some conclusions in Section 6.

2 Models of Services

In this section, we focus on formal models. Bringing formality to the service development process opens the way to the writing and verification of properties that the designer expects from his/her system. This is not the case of semi-formal notations such as UML or BPMN which are often acknowledged as more readable and user-friendly than formal methods but lack formal semantics and validation tools. Services are distributed components which communicate exchanging messages, therefore they are best described using behavioural description languages. Several candidates have been used in the literature:

- Process algebras (or calculi): CCS, CSP, LOTOS, FSP, etc
- Automata-based models: state diagrams, Harel's Statecharts, IO-Automata, LTS, etc
- Petri nets: coloured Petri nets, workflow nets, open nets, etc
- Temporal Logic: Lamport's TLA
- Message Sequence Charts

Here are a few references [19, 38, 18, 7, 1] where the reader can find more details about these models and their use in the service development process. According to us, process algebras are one of the best candidates to specify service models for four reasons: (i) the existing calculi present several levels of abstraction useful to have a more faithful representation of a service, *e.g.*, specifying data (LOTOS) or mobility (π-calculus), (ii) they are compositional notations, then adequate to describe composition of services, (iii) they provide textual notation which makes them scalable to tackle real-world systems, and (iv) there exist some state-of-the-art verification tool-boxes for these languages, *e.g.*, SPIN, CADP, UPPAAL, or μ-CRL2.

In the rest of this paper, for illustration purposes and for the sake of readability (process algebras are not perfect, unfortunately), we assume that services are modelled using *Labelled Transition Systems* (LTSs). An LTS 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 $T \subseteq S \times A \times S$ is the transition relation. In our model, a *label* is either a τ (internal action) or a tuple (m,d) where m is the message name, and d stands for the communication direction (either an emission ! or a reception ?). Labels can take typed parameters or arguments into account as well, and in such a case the transition system is called *symbolic* (STS). Using this 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 (internal choice). LTSs and STSs can be easily derived from higher-level description languages such as Abstract BPEL, see for instance [19, 38, 11] where such abstractions were used for verification, composition or adaptation of Web services. The operational semantics of STSs is given in [17].

Several communication models can be assumed among services. In particular, we would like to say a word here about synchronous *vs.* asynchronous communication. Synchronous communication corresponds to handshake communication whereas asynchronous communication uses message queues for interaction purposes (similarly to mailboxes). Most existing works rely on synchronous communication. Asynchronous communication is as realistic as synchronous communication, however, results are more complicated to obtain and even sometimes undecidable [6] (see Section 6 for a more detailed discussion). In this paper, we assume a binary communication model where two services synchronize if one can evolve through an emission, the other through a reception, and both labels share the same message.

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 3 shows two versions of one service protocol without (S1) and with (S1') its internal behaviour. Assuming a synchronous communication model, S1 and S2 can interoperate on a and terminate in final states (b! in S1 has no counterpart in S2 and cannot be executed). 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.

The reader interested in more details about τ transitions and their handling can refer to [33].

3 Automated Verification

A major interest of using abstract languages grounded on a clear semantics is that automated tools can be used to check that a system matches its requirements and operates safely. Specifically, these tools can

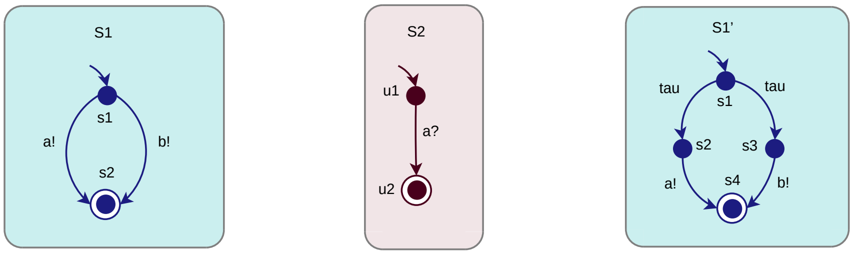

Figure 3: S1 and S2 interoperate successfully, but S1' and S2 can deadlock

help (i) checking that two services are in some precise sense *equivalent* – one behaviour is typically a very abstract one expressing the specification of the problem, while the other is closer to the implementation level; this can also be used for checking the substitutability (or replaceability) of one service by another; (ii) checking that a service (possibly composite) verifies desirable *properties* – *e.g.*, the property that the system will never reach some unexpected state. Revealing that the composition of a number of existing services does not match an abstract specification of what is desired, or that it violates a property which is absolutely needed can be helpful to correct a design or to diagnose bugs in an existing service. Note that in the following of this section, we focus on verification techniques that are of interest for Web services, and we do not give an overview of the many papers that have been published on this topic (most of them do the same using different languages and tools), see for instance [19, 38, 18, 7, 35].

3.1 Verifying Equivalences

Intuitively, two services are considered to be equivalent if they are *indistinguishable* from the viewpoint of an external observer interacting with them. This notion has been formally defined in the process algebra community, and several notions of equivalence have been proposed [30]. Equivalences are strong yet suitable relations for these checks, because they preserve all observable actions. However, these notions exhibit some subtleties relevant to the context of Web services.

A first approach is to consider two services to be equivalent if the set of *traces* they can produce is the same (*trace-equivalence*). For instance, the possible executions of the services shown in Fig. 4 part (A), where messages a, b and c can be respectively understood as requests for reservation, editing data and cancellation. Both of these two services will have a.b and a.c as possible traces: they will either receive the messages a then b, or a then c.

Nevertheless, it is not fully satisfactory to consider these two services equivalent since they exhibit the following subtle difference. After receiving message a, the first service will accept either message b or c. The second service behaves differently: on receiving message a, it will either choose to move to a state where it expects message b, or to a state where it expects message c. Depending on the choice it makes, it will not accept one of the messages whereas the first service leaves both possibilities open. The second service does not guarantee that a request for reservation (a) followed by, *e.g.*, cancellation (c) will be handled correctly (c might not be possible if the service has chosen the left-hand side branch). The notion of equivalence called *bisimulation* [30] is a refinement of trace equivalence which takes these differences into account.

Further subtleties arise when one has a partial knowledge of the service behaviour. This may happen for two reasons: (i) during the design stage, where the specification which is being defined is abstract and incomplete; (ii) when one finds or reuses an existing service, and only an interface or a partial description hiding private details is available. τ actions must be taken into account when reasoning on

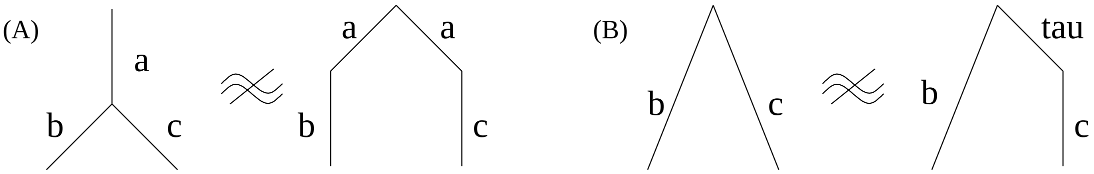

Figure 4: Classical examples of services not observationally equivalent.

the equivalence of two services, as evidenced by Fig. 4 part (B). Both of the services depicted here can receive b (edition of reservation data) or c (cancellation). But whereas the first one can receive any of the two, the second one can choose to first execute some unobservable action which will lead it to a state where it can only receive message c. Once again it cannot be guaranteed that the second service will accept cancellation requests, and this depends on some decisions it takes internally.

Weak (or observational) and branching equivalences are the strongest of the weak equivalences [24], branching equivalence being the strongest of these two. They preserve behavioural properties (do not add deadlocks for instance) on observable actions, and are therefore acknowledged as the most appropriate notions of process equivalence, in the context of Web services. They are implemented in tools like CADP [23] which can automatically check that two transition systems denote the same observational (or branching) behaviour. Another notion called *strong bisimulation* exists. It is nevertheless too restrictive in our context because it imposes a strict matching of the τ actions. Also note the notion of *congruence*, an observational equivalence which should be preferred when one wants services to be equivalent *in any context*, *i.e.*, in all possible systems using them.

3.2 Verifying Properties

The properties of interest in concurrent systems typically involve reasoning on the possible scenarii that the system can go through. An established formalism for expressing such properties is given by *temporal logics*[1] like CTL$\star$ [28]. These logics present constructs allowing to state in a formal way that, for instance, all scenarii will respect some property at every step, or that some particular event will eventually happen, and so on.

An introduction to temporal logic goes beyond the aims of this paper, but it suffices to say that a number of classical properties typically appear as patterns in many applications. Reusing them diminishes the need to learn all subtleties of a new formalism. The most noticeable properties are:

- **Safety properties**, which state that an undesirable situation will never arise. For instance, the requirements can forbid that the system reserves a room without having received the credit information from the bank;
- **Liveness properties**, which state that some actions will always be followed by some reactions; a typical example is to check that every request for a room will be acknowledged.

The techniques used to check whether a system respects temporal logic properties are referred to as *model checking* methods [15]. Several tools exist and can be used to model-check abstract descriptions of services, *e.g.*, CADP, or SPIN.

---

[1] This name should not give the impression that these logics introduce a quantitative notion of time, they are indeed used to express constraints on the possible executions of a system.

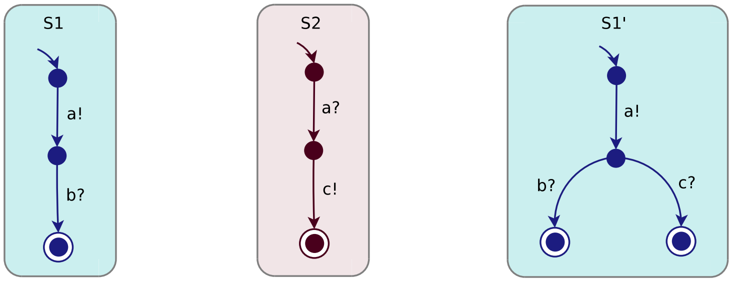

Figure 5: Deadlock-freeness compatibility

4 Compatibility and Adaptation

4.1 Compatibility Notions

Compatibility aims at ensuring that services will be able to interact properly, that is satisfy a specific criterion on observable actions and terminate in final states. Typically, compatibility is needed at design-time as a previous step (discovery) in a service composition construction in order to avoid erroneous executions at run-time. Substitutability is a similar issue and aims at replacing one service by another without introducing flaws. Substitutability can be checked using equivalence-checking techniques presented in the previous section. Compatibility checking, if defined in a formal way, can be automated using state space exploration tools such as CADP or SPIN, or rewriting-based tools such as Maude.

In the rest of this subsection, we introduce three notions of compatibility, namely deadlock-freeness, unidirectional-complementarity and unspecified-receptions, that make sense in the Web services area. These notions have often been studied in the literature [6, 43, 12, 5, 17, 33].

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 5 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 [30] 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 2 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'.

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,

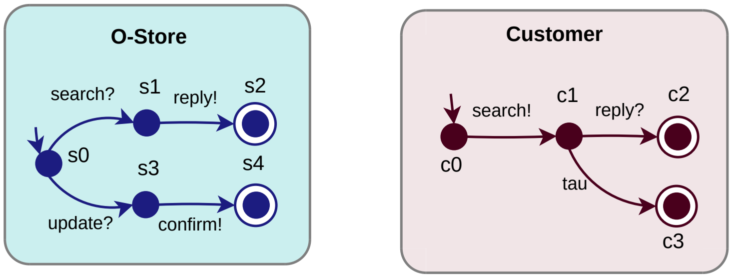

Figure 6: An online store

both services must be free of deadlocks. In Figure 1, 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 [5, 17].

4.2 Compatibility Degree

Most of the approaches existing for checking compatibility return a “True” or “False” result to detect whether services are compatible or not. Unfortunately, a Boolean answer is not very helpful for many reasons. First, in real world case studies, there will seldom be a perfect match, and when service protocols are not compatible, it is useful to differentiate between services that are slightly incompatible and those that are totally incompatible. Furthermore, a Boolean result does not give a detailed measure of which parts of service protocols are compatible or not. To overcome the aforementioned limits, a new solution aims at measuring the compatibility degree (or similarity degree if the idea is to replace and not to compose services) of service interfaces. This issue has been addressed by a few recent works, see for instance [40, 32, 27, 2, 42, 34].

Let us illustrate with a simple example (Fig. 6) the kind of results one can compute with these compatibility measuring approaches. Here, we use the compatibility measuring algorithms presented in [34]. This approach takes as input two STSs and computes a compatibility degree for each global state, *i.e.*, each couple of states (s_i, s_j) with $s_i \in S_1$ and $s_j \in S_2$. All compatibility scores range between 0 and 1, where 1 means a perfect compatibility. To measure the compatibility of two service protocols, the protocol compatibility degrees are computed for all possible global states using a set of static compatibility measures. This work uses three static compatibility measures, namely state natures, labels, and exchanged parameters. These measures are used next to analyse the behavioural part (ordering of labels) of both protocols. Intuitively, two states are compatible if their backward and forward neighbouring states are compatible, where the backward and forward neigbours of state s' in transitions (s, l, s') and (s', l', s'') are respectively the states s and s''. Hence, in order to measure the compatibility degree of two service protocols, an iterative approach is considered which propagates the compatibility degree from one state to all its neighbours. This process is called compatibility flooding.

Table 1 shows the matrix computed for the example depicted in Figure 6 according to the unidirectional-complementarity notion. Let us comment the compatibility of states c0 and s0. The measure is quite high because both states are initial and the emission search! at c0 perfectly matches the reception search? at s0. However, the compatibility degree is less than 1 due to the backward propagation of the deadlock from the global state $(\mathsf{s1}, \mathsf{c3})$ to $(\mathsf{s1}, \mathsf{c1})$, and then from $(\mathsf{s1}, \mathsf{c1})$ to $(\mathsf{s0}, \mathsf{c0})$.

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

Table 1: The compatibility matrix computed for the example in Figure 6

4.3 Service Adaptation

While searching a service satisfying some specific requirements, one can find a candidate which exhibits the expected functionality but whose interface does not exactly fit in the rest of the system. *Software Adaptation* [3] is a very promising solution to compose in a non-intrusive way black-box components or (Web) services although they present interface mismatches. Adaptation techniques aim at automatically generating new components called *adaptors*, and usually rely on an *adaptation contract* which is an abstract description of how mismatches can be worked out. All the messages pass through the adaptor which acts as an orchestrator, and makes the involved services work correctly together by compensating mismatches. The generation of this adaptor is a complicated task, especially when interfaces take into account a behavioural description of the service execution flow. Recently, several approaches have been proposed to generate service adaptors, see for example [8, 31, 29, 13, 1].

Figure 7 gives an example: the first interface corresponds to an SQL service which can receive (req?) and answer (result!) requests, stops (halt!), or halts temporarily for maintenance purposes (maintenance? and activation?). The client can submit requests (request!), and receive responses (request?).

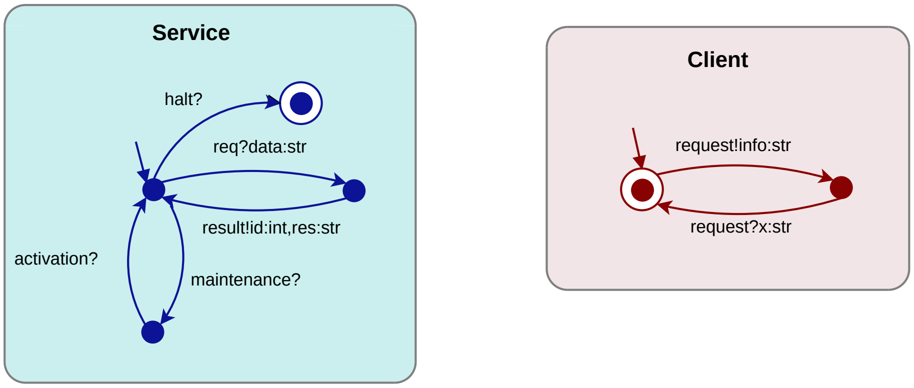

Figure 7: An SQL service

Several notations exist for writing adaptation contracts. In this paper, we use *vectors* [29] which specify interactions between several services. They express correspondences between messages, like bindings between ports, or connectors in architectural descriptions. Each label appearing in one vector is executed by one service and the overall result corresponds to an interaction between all the involved services. Furthermore, variables are used as placeholders in message parameters. The same variable name appearing in different labels (possibly in different vectors) enables one to relate sent and received arguments of messages.

As far as our example is concerned, the following vectors constitute a contract from which the adaptor protocol given in Figure 8 is automatically generated by using techniques and tools presented in [29].

This approach respectively generates (i) LOTOS code[2] for service interfaces and the contract, and (ii) the corresponding state space by applying on-the-fly simplification (deadlock suppression) and reduction techniques (τ transition removal).

$V1 = \langle s\!:\!req?X; c\!:\!request!X\rangle \qquad V2 = \langle s\!:\!result!Y,Z; c\!:\!request?Z\rangle \qquad V3 = \langle s\!:\!halt?\rangle$

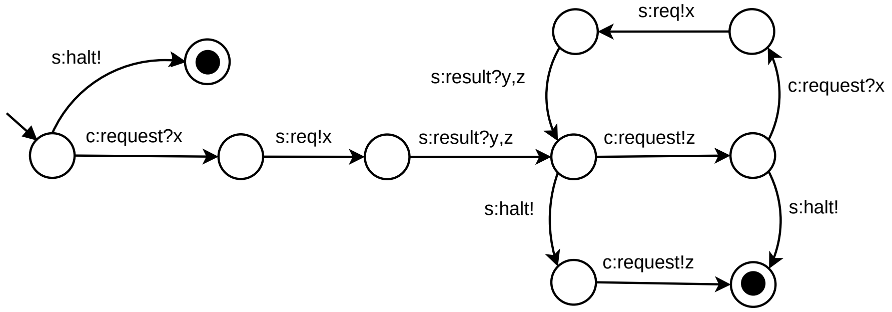

Figure 8: The adaptor protocol for the SQL example

From adaptor protocols, either a central adaptor can be implemented, or several service wrappers can be generated to distribute the adaptation. In the former case, the implementation of executable adaptors from adaptor protocols can be achieved for instance using techniques presented in [29] and [16] for BPEL and Windows Workflow Foundation, respectively. In the latter case, each wrapper constrains its service functionality to make it respect the adaptation contract [37].

5 Realizability and Conformance

Interactions among a set of services involved in a new system can be described from a global point of view using *choreography* specification languages. Several formalisms have already been proposed to specify choreographies: WS-CDL, collaboration diagrams, process calculi, BPMN, SRML, etc. Given a choreography specification, it would be desirable if the local implementations, namely *peers*, can be automatically generated via projection. However, generation of peers that precisely implement the choreography specification is not always possible: This problem is known as *realizability*. A related problem is known as *conformance* where the question is to check whether a choreography and a set of service implementations (not obtained by projection from the choreography) produce the same executions.

A couple of unrealizable collaboration diagrams [9] are presented in Figure 9. The first one (left hand side) is unrealizable because it is impossible for the peer `C` to know when the peer `A` sends its `request` message since there is no interaction between `A` and `C`. Hence, the peers cannot respect the execution order of messages as specified in the collaboration diagram. The second one is slightly more subtle because this diagram is realizable for synchronous communication, and unrealizable for asynchronous communication. Indeed, in case of synchronous communication, the peer `C` can synchronize (rendezvous) with the peer `A` only after the `request` message is sent, so the message order is respected. This is not the case for asynchronous communication since `A` cannot block `C` from sending the `update` message. Hence, `C` has to send the `update` message to `A` without knowing if `A` has sent the `request` message or not. Therefore, the correct order between the two messages cannot be satisfied. We also show in Figure 9 the LTS generated for peer `A` by projection.

---

[2] LOTOS is a value passing process algebra proposed in the late 80s, see [4] for more details.

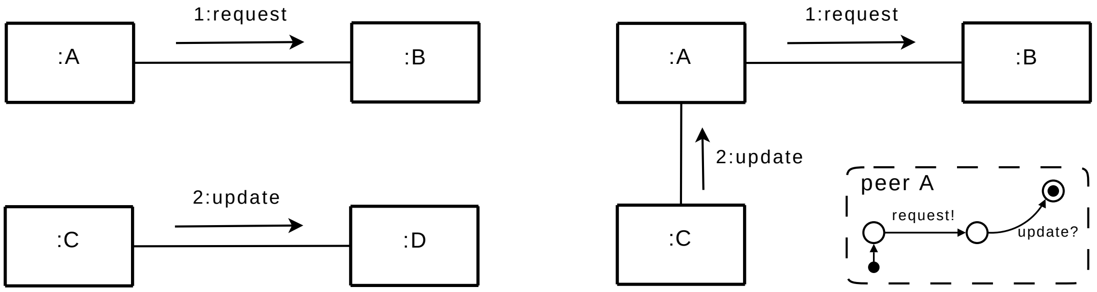

Figure 9: Examples of unrealizable collaboration diagrams

Several works aimed at studying and defining the realizability (and conformance) problem for choreography, here are a few references [25, 10, 26, 22, 9]. In [10, 26], the authors define models for choreography and orchestration, and formalize a conformance relation between both models. Other works [14, 36] propose well-formedness rules to enforce the specification to be realizable. A few works [36, 39] also propose to add messages in order to implement unrealizable choreographies. Fu *et al.* [20] proposed three conditions (lossless join, synchronous compatible, autonomous) that guarantee a realizable conversation protocol under asynchronous communication. These conditions have been implemented in the WSAT tool [21] which takes a conversation protocol as input, and says if it satisfies the three realizability conditions. [41] discusses some interesting open issues in this area.

6 Concluding Remarks

In this paper, we have surveyed some issues in Web services which require analysis and verification techniques. Using these techniques seems natural when one wants to ensure that a composition of services will work correctly or satisfy some high-level requirements. But they have also other applications in SOC, *e.g.*, to check the compatibility of a service with a possible client (discovery), or to generate some service adaptors if some interface mismatches prevent their direct composition. Last, we have showed that when specifying a system using choreography languages, some analysis are useful to check that the corresponding distributed implementation will behave as described in the global specification.

We would like to conclude with a few challenges which are still some open issues, as far as analysis techniques are concerned, in the Web services domain. All these challenges assume an asynchronous communication model (that is based on message queues). A few works already exist, in [22] for example the authors define a synchronizability condition which makes systems under asynchronous communication verifyable with tools working with synchronous comunication. Some sufficient conditions have also been proposed to guarantee the realizability of conversation protocols [20]. Nevertheless, in both works, if these conditions are not satisfied, nothing can be concluded on the system being analysed.

Some open challenges assuming an asynchronous communication model are the following: (i) providing automated techniques to check the compatibility of two or more services, (ii) checking the adaptability of a set of services being given an adaptation contract, and if the system is adaptable, generating the corresponding adaptor, (iii) finding a decidable algorithm for checking the realizability of a choreography specification language with loops (such as conversation protocols [20]).

Acknowledgements. The author would like to thank Meriem Ouederni for her comments on a former version of this paper.

References

[1] W. M. P. van der Aalst, A. J. Mooij, C. Stahl & K. Wolf (2009): *Service Interaction: Patterns, Formalization, and Analysis*. In: *Proc. of SFM'09*, *LNCS* 5569, Springer, pp. 42–88.

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

[3] 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*, *LNCS* 3938, Springer, pp. 193–215.

[4] T. Bolognesi & E. Brinksma (1987): *Introduction to the ISO Specification Language LOTOS*. *Computer Networks* 14, pp. 25–59.

[5] L. Bordeaux, G. Salaün, D. Berardi & M. Mecella (2004): *When are Two Web Services Compatible?* In: *Proc. of TES'04*, *LNCS* 3324, Springer, pp. 15–28.

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

[7] F. van Breugel & M. Koshkina (2006): *Models and Verification of BPEL*. Technical Report.

[8] A. Brogi & R. Popescu (2006): *Automated Generation of BPEL Adapters*. In: *Proc. of ICSOC'06*, *LNCS* 4294, Springer, pp. 27–39.

[9] T. Bultan & X. Fu (2008): *Specification of Realizable Service Conversations using Collaboration Diagrams*. *Service Oriented Computing and Applications* 2(1), pp. 27–39.

[10] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi & G. Zavattaro (2006): *Choreography and Orchestration Conformance for System Design*. In: *Proc. of Coordination'06*, *LNCS* 4038, Springer, pp. 63–81.

[11] 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.

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

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

[14] M. Carbone, K. Honda & N. Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *Proc. of ESOP'07*, *LNCS* 4421, Springer, pp. 2–17.

[15] E. M. Clarke, O. Grumberg & D. Peled (2000): *Model Checking*. The MIT Press.

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

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

[20] X. Fu, T. Bultan & J. Su (2004): *Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services*. *Theor. Comput. Sci.* 328(1-2), pp. 19–37.

[21] X. Fu, T. Bultan & J. Su (2004): *WSAT: A Tool for Formal Analysis of Web Services*. In: *Proc. of CAV'04*, *LNCS* 3114, Springer, pp. 510–514.

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

[23] 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*, *LNCS* 4590, Springer, pp. 158–163.

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

[25] R. Kazhamiakin & M. Pistore (2006): *Analysis of Realizability Conditions for Web Service Choreographies*. In: *Proc. of FORTE'06*, *LNCS* 4229, Springer, pp. 61–76.

[26] J. Li, H. Zhu & G. Pu (2007): *Conformance Validation between Choreography and Orchestration*. In: *Proc. of TASE'07*, IEEE Computer Society, pp. 473–482.

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

[28] Z. Manna & A. Pnueli (1995): *Temporal Verification of Reactive Systems: Safety*. Springer.

[29] 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*, *LNCS* 5364, Springer, pp. 84–99.

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

[31] H. R. Motahari-Nezhad, B. Benatallah, A. Martens, F. Curbera & F. Casati (2007): *Semi-Automated Adaptation of Service Interactions*. In: *Proc. of WWW'07*, pp. 993–1002.

[32] S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook & P. Zave (2007): *Matching and Merging of Statecharts Specifications*. In: *Proc. of ICSE'07*, ACM Press, pp. 54–64.

[33] M. Ouederni & G. Salaün (2010): *Tau Be or not Tau Be? A Perspective on Service Compatibility and Substitutability*. In: *Proc. of WCSI'10*, EPTCS.

[34] M. Ouederni, G. Salaün & E. Pimentel (2010): *Quantifying Service Compatibility: A Step Beyond the Boolean Approaches*. In: *Proc. of ICSOC'10*, Springer.

[35] M. E. Cambronero Piqueras, G. Dias Descalzo, V. Valero Ruiz & E. Martinez Lopez (2010): *Validation and Verification of Web Services Choreographies by Using Timed Automata*. *JLAP* .

[36] Z. Qiu, X. Zhao, C. Cai & H. Yang (2007): *Towards the Theoretical Foundation of Choreography*. In: *Proc. of WWW'07*, ACM Press, pp. 973–982.

[37] G. Salaün (2008): *Generation of Service Wrapper Protocols from Choreography Specifications*. In: *Proc. of SEFM'08*, IEEE Computer Society, pp. 313–322.

[38] 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.

[39] G. Salaün & T. Bultan (2009): *Realizability of Choreographies using Process Algebra Encodings*. In: *Proc. of IFM'2009*, *LNCS* 5423, Springer, pp. 167–182.

[40] O. Sokolsky, S. Kannan & I. Lee (2006): *Simulation-Based Graph Similarity*. In: *Proc. of TACAS'06*, *LNCS* 3920, Springer, pp. 426–440.

[41] J. Su, T. Bultan, X. Fu & X. Zhao (2007): *Towards a Theory of Web Service Choreographies*. In: *Proc. of WS-FM'07*, *LNCS* 4937, Springer, pp. 1–16.

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

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