Reconciling Web service failing interactions. Toward an approach based on automatic generation of mediators

Ali Ait-Bachir, Marie-Christine Fauvet

Reconciling Web service failing interactions. Towards an approach based on automatic generation of mediators

Ali Aït-Bachir, Marie-Christine Fauvet
University of Grenoble, LIG (MRIM)
385 rue de la bibliothèque – B.P. 53 – 38041 Grenoble Cedex 9, France
Ali.Ait-Bachir@imag.fr, Marie-Christine.Fauvet@imag.fr

Abstract

Interactions between Web services are based on interfaces which describe Web services on both structural and behavioural perspectives. It can happen that the interface provided by a service does no longer match (for instance, because of an evolution) the interface required by its partners. In this situation, and until the required interfaces are fixed, interactions cannot succeed. To address this issue, and focusing on the behavioural part of interfaces, we propose an approach based on a mediator, automatically generated, which aims to seamlessly resolve incompatibilities during service interactions.

1 Introduction

As Web service interactions rely on message exchanges, modelling Web service aims at describing messages as well from the structural point of view (types of exchanged messages) as from the behavioural point of view (control flow between message exchanges). With this setting, a Web service's *interface* is defined as the set of messages it can receive and send, and the inter-dependencies between these messages. We distinguish the *provided* interface an existing service exposes, from its *required* interface as it is expected by its clients (i.e. applications).

As a Web service evolves, its interface is more likely to be modified too. This leads to the situation where the provided interface of a service does no longer correspond to the one its partners expect. Two solutions thus apply: (1) modify the service in order to make the interface it provides match the interface required by each client; (2) introduce an adapter that reconciles the provided interface with those required by the partners. The former solution is not satisfying because the same service may interact with many other partners which consider its original interface. The same service has to expose as many provided interfaces as collaborations it is involved in. The latter solution consists in supplying an adapter which is capable of matchmaking each of the required interfaces with the one provided by the service.

A service is generally described according to its structural or behavioural dimensions, or even according to its non-functional dimension. Thus, interface matchmaking must be studied according each of these dimensions. Dealing with structural matchmaking essentially leads towards reconciliation between different message types. This issue has been widely studied (see for example [13, 11]) and many commercial systems exist (e.g. Microsoft's BizTalk Mapper). Conversely the problem of behavioural matchmaking is still a research topic [12, 1, 2].

The study reported in the paper aims at dealing with this latter issue. Its main contribution is the definition and implementation of an architecture for service providers whose main modules are:

- *Mediators* (one associated to each service). A mediator is responsible for accepting messages received (on the behalf of the service), detecting incompatibilities, and if necessary, seamlessly reconciling failing interactions.
- *A generator* of mediators. Mediators are implemented by finite state automaton each of which automatically generated. The generation process, according to the previous version of the provided interface and its current one issued by the last modification detects: (1) whether the current version simulates the previous one and if not (2) generates the mediator. As a first stage, the mediator is limited to consider deletion of operations only.

The rest of the text is structured as follows. In Section 2 we frame the problem addressed and discuss related work. In section 3, we describe the main principles of our approach. The processes we propose for incompatibility detection and mediator generation are detailed in Section 4. While Section 5 exposes implementation details, Section 6 concludes and sketches further work.

[0]This work has been partially funded by the project Web Intelligence granted by the French Rhône- Alps Region.

2 Motivation and Related Work

To illustrate the issues we deal with, we consider a service meant to sell and deliver goods. In Figure 1, an activity diagram (UML) is given which models interactions the service BestBargain is capable to handle, and their inter dependencies. The activity is named according to the type of message being sent or received.

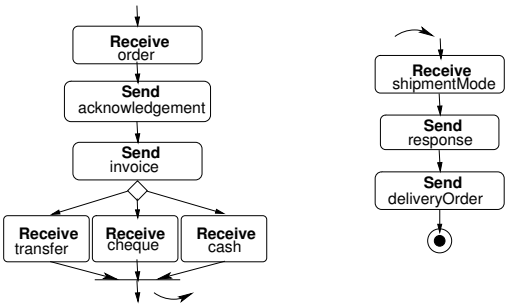

Figure 1. BestBargain's provided interface.

As said before, we focus only on the behavioural dimension of interfaces. With this setting, a modification could be either an addition, a deletion, or an update of an activity, or any combination of these modifications. Deletion of an operation is studied according to its nature:

- The operation to be deleted belongs to a sequence of operations. Such an example could be the deletion of the activity Send response from BestBargain's interface. After this modification, clients whose required interface still contains the corresponding operation (an incoming message), will expect the message but never receive it.
- The operation to be deleted corresponds to a choice in a conditional composition. For instance, the activity Receive cash is no longer available. Thus, clients whose interfaces still consider this option and which are willing to use it can no longer send it as the service does not expose any more the operation for its reception.

The mediator-based approach proposed in this paper aims at detecting interactions which fail because of incompatibilities between the interface a service provides and the one its clients require. In the first of the situations discussed above, the mediator, to allow the conversation to carry out, sends, when necessary, a message of the same type of Send response, so the client gets an incoming message, as close as possible to what it was expecting. In the second situation, the mediator intercepts the message Receive cash sent by the client, and on the behalf of the service returns a message asking for another choice.

The issues illustrated below have been partially addressed before, with various points of view. Web service interactions may fail because of interface incompatibilities according to their structural dimension. In this context, reconciling incompatible interactions leads towards transforming message types (using for instance Xpath, XQuery, XSLT). Issues that arise in this context are similar to those widely studied in the data integration area. A mediation-based approach is proposed in [1]. While this approach relies, as ours, on a mediator (called *virtual supplier*) it focuses on structural dimension of interfaces only. The virtual supplier is responsible for adapting the structure of exchanged messages.

Conformity test of interfaces has been widely studied in the context of Web service composition. Most of approaches which focus on the behavioural dimension of interfaces rely on equivalence and similarity calculus to check, *at design time*, whether interfaces, described for instance by automata, are compatible (see for example [8, 3, 9]). The behavioral interrface describes the structured activities of a business process. Some studies discuss the translation into automaton, of interfaces described in WSCI[1] or BPEL[2]. Petri nets may also be used to formalise interface's behavioural dimension. Checking interface compatibility is thus based on bi-similarity algorithms [5, 10]. These approaches do not deal with reconciliation issues when incompatibilities occur.

Recent research has addressed interface adaptation issues. In [2], authors present a framework which relies on identifying *mismatch patterns*. Other techniques for generating adaptors are defined in [14, 6]. In [6], authors propose an algebra of interface transformation operators and a visual language based on this algebra. All these studies differ from ours in that they provide means for developers to implement adaptors, while we aim at automatising the generation of mediators.

3 Preliminaries

In this section we first introduce the mediation principle which is behind our approach, then we show how we model services' behaviour with automata. Eventually, we introduce notations and give definitions necessary for the understanding of our solution.

Mediation principles: the cornerstone of the conversation mediation between two Web services, is that of each message being sent or received by the two partners, pass through a third partner called a *mediator* [1]. For instance, a message sent by a client is intercepted first by the mediator which checks whether the message still conforms to the service's interface. If yes, the message is forwarded to its recipient, otherwise the mediator tries to reconcile the resulting incompatibilities. In our approach, the mediator

[1] Web Service Choreography Interface [12].

[2] Business Process Execution Language [10]

is automatically generated according to the version of the interface issued by the last evolution and its previous one.

We choose to model the behaviour of a Web service with *Labelled Transition Systems (LTS)*[3]. A LTS is a directed graph where nodes refer to all possible states of a system and arcs model transitions between states [9]. Each arc is labelled with an event. A guard condition could be associated to transition.

In our context of modelling behavioural dimension of interfaces, transitions are labelled with messages to be exchanged. When a message is sent or received, the corresponding transition is fired (it is so, only when the guard, if any, is evaluated to true). Internal operations, which are not shown in services' interface, are also modelled by transitions. Each state describes a particular step of the business process implemented by the service. Operations on data can also be defined within states (e.g. to update the value of variables associated to the process instance's context).

Figure 2 depicts the LTS modelling the shipment phase of the illustrating example presented in Section 2. In this example, after the information about the payment is sent (by transfer, cheque or cash), the client sends his preference for the shipment (express, the day after, etc.). The operation enacted by this message, sends in turn, a response towards the client. Then, the delivery order is issued by the service and sent to the client.

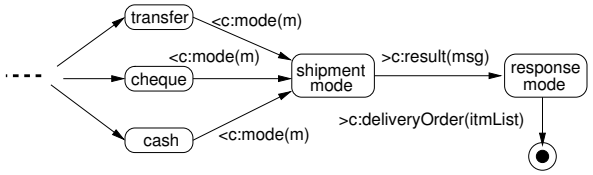

Figure 2. Shipment mode LTS

The message m is denoted by $>m$ (respectively $<m$) when it is sent (respectively received) (see Figure 2). Each conversation initiated by a client (identified by c) generates an instance of the corresponding automaton.

Definitions and Notations: A LTS is a tuple (S, L, T, s_0, F) where: S is a set of states, L a set of events (actions), T the transition function. s_0 is the initial state such as $s_0 \in S$, and F the set of final states such as $F \subset S$. The transition function T associates a source state $s_1 \in S$ and an event $l_1 \in L$ to a target state $s_2 \in S$.

We denote by P' the version of the interface issued by the last modification while P denotes the previous one. Both provided interfaces, P and P', are respectively described by the following LTSs: (S, T, s_0, F, L), and (S', T', s'_0, F', L').

[3]Petri Net-based was another option. We do not discuss this choice here, as it is out of the scope of this paper.

We adopt the following notations [5]. Examples refer to the LTS depicted in Figure 2:

- $s\bullet$ is the set of outgoing transitions from s (e.g. shipmentMode$\bullet$ = { >c:result(msg)}).
- $\bullet s$ is the set of the incoming transitions for the state s (e.g. $\bullet$responseMode= {>c:result(msg)}).
- $t\circ$ is the target state of the transition t (e.g. >c:result(msg)$\circ$ =responseMode).
- $\circ t$ is the source state of the transition t (e.g. $\circ$>c:result(msg)=shipmentMode).

The $\circ$ operator (respectively $\bullet$) is generalised to a set of transitions (respectively states). For example, if T is a set of transitions such as: $T = \bigcup_{i=1}^{n}\{t_i\}$ then $T\circ = \bigcup_{i=1}^{n}\{t_i\circ\}$.

Compatibility checking between two interfaces relies on simulation calculus [4]. Communication operations (named distinguishable behaviour, in [3]) only are considered in this process. To abstract internal operations in an automaton, we use a reduction algorithm which eliminates the $\epsilon - transitions$ [15]. This algorithm is applied to LTSs whose internal operations are considered as $\epsilon - transitions$.

To check whether a mediator is needed, it is necessary to identify situations when the version P' does not simulate its previous one P. If P' simulates P (denoted $P \preceq P'$) then each interface R required by a client, which is compatible with P (denoted $R \sim P$) remains compatible with P' as shown below. Let $\overline{R}$ denotes the opposite interface of R obtained by transforming each message being sent to a message being received and conversely. Given $R \sim P$, thus $\overline{R} \preceq P$ [3]. Due to the transitivity property of the preorder relation of simulation [4], we have $\overline{R} \preceq P'$ (because $\overline{R} \preceq P$ and $P \preceq P'$). Hence, R conforms to P' ($R \sim P'$). Thus, the detection and the resolution of incompatibilities are relevant only if P' does not simulate P.

4 Detection and Resolution

In the study reported in this text, we focus on incompatibilities due to the deletion of operations (i.e. deletion of messages to be sent or received as specified in the provided interface). Deleting operations is discussed depending on three cases: deleting an operation or a sequence of operations (see Section 4.1). We distinguish the specific case of an acknowledgement (see Section 4.2); the third case we consider consists in the deletion of an option among a conditional composition (see Section 4.3).

To detect incompatibilities, P and P' are browsed in parallel from their respective initial states s_0 and s'_0. The search seeks for two states s and s' (belonging respectively to P and P') which are such as the sub-automaton starting from s in P and the one starting from s' in P' are *incompatible*. This search is implemented by functions detailed below. Each returns a tuple $< typeI, s, s' >$ where $typeI$ is

the type of the incompatibility between the sub-automaton starting respectively from s and s'. These functions are called by the algorithm presented in Section 5.

4.1 Deleting an operation from a sequence

As illustrated in Section 2, after an operation (or a sequence of operations) has been deleted from the service's provided interface, clients can no longer use it. To reconcile the resulting failing interactions, the mediator simulates the existence of this operation whose corresponding message is not forwarded to the service. The mediators's behaviour is defined in such a way it can handle the reception of such messages as well as those which have to be forwarded to the service because they are exposed in the current version of its interface (see Figure 3).

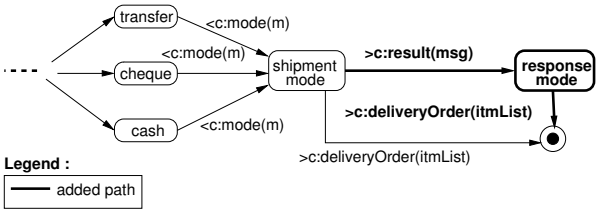

Figure 3. Mediator's LTS (deletion of an operation)

With respect to our scenario, the version P of the provider contains an operation that permits clients to choose the shipment mode (see Figure 2). The corresponding operation is followed by an other one whose role is to return the shipment mode which will eventually be chosen (it could be different than the one wished by the client). After the evolution which produced the new interface P', this operation is no longer available. The automaton shown in Figure 3, is derived from the one associated to P where the state responseMode and both transitions >c:result(msg) and >c:deliveryOrder(itmList) have been added (see the path in bold, shown in Figure 3). The message associated to >c:result(msg) conforming to its type defined in P is sent to clients which require P, and wait for an answer before they can carry out and accept the message >c:deliveryOrder(itmList).

The detection of this kind of evolution is formalised by the following expression. An operation is defined in P but missing in P', if it exists a pair s and s' (respectively in P and P') which satisfies:

$s\bullet \neq \emptyset \wedge$

$(s'\bullet = \emptyset \vee (s'\bullet \neq \emptyset \wedge s\bullet \cap s'\bullet = \emptyset \wedge (((s\bullet)\circ)\bullet) \subseteq s'\bullet))$

The deleted operation is detected when s has outgoing transitions (i.e. s is not a final state, expressed by $s\bullet \neq \emptyset$) and s' does not have any outgoing transitions (s' is a final state, formulated by $s'\bullet = \emptyset$). An other case of operation deletion is detected when the outgoing transitions of s are not included in the outgoing transitions of s' (formulated by $s\bullet \cap s'\bullet = \emptyset$). In this case, the outgoing transitions of the subsequent states of s are included in the outgoing transitions of s' (formulated by $(((s\bullet)\circ)\bullet) \subseteq s'\bullet$).

The expression above can be generalised in order to detect the deletion of a sequence of operations:

$s\bullet \neq \emptyset \wedge$

$(s'\bullet = \emptyset \vee (s'\bullet \neq \emptyset \wedge s\bullet \cap s'\bullet = \emptyset \wedge\ IncEt(s\bullet, s'\bullet))$

where: $IncEt(s\bullet, s'\bullet) =$

$s\bullet \neq \emptyset \wedge s\bullet \subseteq s'\bullet \wedge \bigcup_{i=1}^{\|(s\bullet)\circ\|} IncEt((((s\bullet)\circ)_i)\bullet, s'\bullet)$

The idea is to check whether it exists at least one subsequent state of s whose outgoing transitions are included in the set of outgoing transitions of s'. This test is based on the function $IncEt(s\bullet, s'\bullet)$ which implements an extended inclusion. If such incompatibility is detected for a pair $< s, s' >$, a path, containing a sequence of transitions and states, is built starting from s in P and ending in a state whose outgoing transitions are included in $s'\bullet$. The mediator's automaton is first built by the one describing P'. Then, this path is added to it, starting from s' and ending in states $(s'\bullet)\circ$.

4.2 Deletion of an Acknowledgement

After deleting an acknowledgement, clients who are unaware of these changes may wait indefinitely for this message that will never come. Therefore, the mediator must be able to act on the behalf of the service, and send a message the client is waiting for.

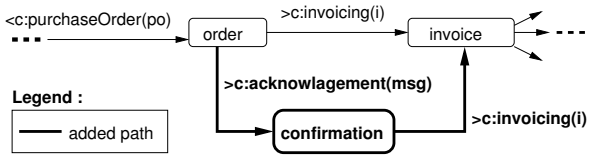

Figure 4. Mediator's LTS (deletion of an acknowledgement)

The LTS modelling the mediator is shown in Figure 4. The mediator is first initialised by the LTS of the new version of the interface. Then, between states order and invoice a path is added (with states confirmation and both transitions >c:send acknowledgement and >c:send invoice). This path is only meant to be used in case a client still requires the previous version of the service's interface.

The expression to detect an acknowledgement deletion is based on the conjunction of the two following boolean expressions (s and s' are states in P and P' respectively):

$1 : \|s\bullet\| = 1 \wedge\ Message(\bullet s) = Message(\bullet s') = \{< m\}$

$2 : Message(s\bullet) = \{> m\} \wedge s\bullet \nsubseteq s'\bullet \wedge ((s\bullet)\circ)\bullet \subseteq s'\bullet$

When a message is received before entering in s and s' (which is expressed by $Message(\bullet s) = Message(\bullet s') = \{< m\}$), the confirmation being sent belongs to $s\bullet$ but it does not in $s'\bullet$ (expressed by $Message(s\bullet) = \{> m\} \wedge s\bullet \not\subseteq s'\bullet$). The confirmation is sent in one of the outgoing transitions of s. Then, the outgoing transitions of the subsequent state of s are included in $s'\bullet$ (formulated by $((s\bullet)\circ)\bullet \subseteq s'\bullet$).

The resolution of this incompatibility is similar to the one which applies for operation deletion, seen before. Thus, starting from s and s', the mediator interface is such as it has a path containing a transition labelled by the acknowledgement operation followed by the next operation after the acknowledgement. The transition of the acknowledgement operation is added within the outgoing transitions of s' and the transition of the operation following the acknowledgement is added within the incoming transitions of the successor state of s' (see Figure 4).

4.3 Deletion of an option in a switch case

To reconcile the conversations which fail because an option has been deleted from an interface, the mediator intercepts the message that contains the option chosen by the client, but not available any more. Instead of forwarding the message to the service, a reject is returned by the mediator to the client, which can, in turn, choose another option. This leads to a mediator whose behaviour is depicted in Figure 5: clients wishing to pay by cash, will receive a reject until they choose another option.

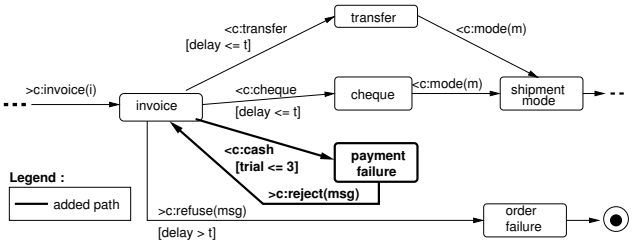

Figure 5. Mediator's LTS (deletion of an option)

The detection of this incompatibility is formalised by the expression below:

$\|Receivings(s\bullet)\| \geqslant 2 \wedge \|s \bullet -s' \bullet \| \geq 1$

where $s \bullet -s'\bullet = \{t \in s \bullet | t \notin s'\bullet\}$

The detection of option deletions aims at verifying first, if the set of the outgoing transitions of s contains at least two transitions associated to message receptions (formulated by $\|Receivings(s\bullet)\| \geqslant 2$), and, if at least one of the outgoing transitions does not exist anymore ($\|s \bullet -s' \bullet \| \geq 1$). For each option that no longer exists in the new version of the interface, a corresponding path is added in the mediator's LTS. Each of these paths starts from a state modelling the choice of the missing option and loops back to its preceding state. A guard is also added to this path in order to limit the number of fails allowed (see Figure 5).

5 Algorithm of incompatibility detection

The algorithm implementing the detection is detailed in Figure 6. Given P' the new version of the service's interface and P the previous one, the principle of the algorithm is to browse in parallel both LTSs describing P' and P. It proceeds so, starting from s_0 and s'_0 the initial states of P' and P LTSs respectively. The algorithm aims at finding all pairs of states s and s' which satisfies one of the three functions Opt, Seq and Ack (each of which associated to each case discussed in the previous section). Each function returns the type of the detected incompatibility (if any) and the pair of states indicating, where in P' and P LTSs, the incompatibility occurs (see line 0).

```
0 type Res = <typeI: {Opt, Seq, Ack}, s1: State, s2: State>
1 Detection (s : State ; P : LTS ; s' : State ; P' : LTS ) :
  { Res }
2 setRes : {Res}, setRes ← ∅                    { result variable }
3 P1,P2 : LTS                                   { intermediary variables }
4 If s• ≠ ∅                                     { condition for no recursive call }
5    setRes ← setRes ∪ Opt(s, P, s', P')
         ∪ Seq(s, P, s', P') ∪ Ack(s, P, s', P')
6    If Opt(s, P, s', P') ≠ ∅                   { deletion of an option }
7       For all t ∈ s• ∩ s'•
8          P1 ← endLTS(P, t○) ; P2 ← endLTS(P', t○)
9          setRes ← setRes ∪ Detection(t○, P1, t○, P2)
10   ElseIf Seq(s, P, s', P') ≠ ∅ or Ack(s, P, s', P') ≠ ∅
11 { deletion of an acknowledgement or operations }
12      P1 ← endLTS(P, s')
13      setRes ← setRes ∪ Detection(s', P1, s', P')
14   Else                                       { No compatibility detected }
15      For all t ∈ s•
16         P1 ← endLTS(P, t○) ; P2 ← endLTS(P', t○)
17         setRes ← setRes ∪ Detection(t○, P1, t○, P2)
18 Return(setRes)
```

Figure 6. Algorithm of incompatibility detection

The initial call of the algorithm is realised by the expression $Detection(s, P, s', P')$. Values for the parameters, at each recursive call, are determined according to the type of deletion, as detected by the functions Opt, Seq and Ack. A function named $endLTS(P: LTS, S: State)$: LTS is introduced. Given an LTS P and one of its state S, $endLTS(P, S)$ is the fragment (an LTS) of P starting from the state S.

If each function (Opt, Seq, and Acq) returns a negative result (no incompatibity detected), then the algorithm is recursively applied on each pair $endLTS(P, sSuc)$, and $endLST(P', sSuc')$ where, for each $t \in s\bullet$, $sSuc = t\circ$ (in P), and $sSuc' = t\circ$ (in P'). Each pair of states $< sSuc, sSuc' >$ is built by applying the same outgoing transition t respectively on s and s' ($s\bullet = s'\bullet$, P and P' are compatible). See lines 15, 16 and 17.

If an incompatibility is detected, the pairs of LTSs to consider next depend on which type of incompatibility is returned. In case of a deletion of an option (see lines 6, 7, 8 and 9), the transitions to be applied are those, in both LTSs, starting from s and s' (i.e. $s \bullet \cap s'\bullet$). In case an operation has been deleted from a sequence, the algorithm goes a step forward in the LTS of P only. The pair of LTSs to consider next is then $endLTS(P, sSuc)$ and P', such as $sSuc\bullet \subseteq s'\bullet$ (see lines 10, 12 and 13). The same process applies in case of the deletion of an acknowledgement.

6 Conclusion and Future Work

In this paper, we focused on conversations which cannot successfully complete because client and service interfaces do not match due to the evolution of the service interface. In our approach, the mediation is based on LTSs (*Labelled Transition Systems*) to model the behavioural dimension of services and to reconcile failing conversations. Three cases of incompatibilities are detected and resolved: deletion of an operation from a sequence, deletion of an acknowledgement and deletion of an option in a conditional composition. The detection and the resolution of incompatibilities is relevant only if the new version P' of the provided interface does not simulate the previous version P. Starting from the detected incompatibilities, the behavioural interface of mediator is generated according to both automata modelling P and P'.

The study reported in this paper raises several research directions. First, we focused on the situation that occurs when a service interface evolves after the substitution of an operation by another new one(adding operation has been addressed in [7]). We also plan to study the completeness of our algorithm for the detection. An other important extension is to generalise the proposed approach, limited to two consecutive versions, and to consider a history of any number of versions.

References

[1] M. Altenhofen, E. Boerger, and J. Lemcke. An execution semantics for mediation patterns. In *Proc. of the BPM'2005 Workshops: Workshop on Choreography and Orchestration for Business Process Managament*, Nancy, France, September 2005.

[2] B. Benatallah, F. Casati, D. Grigori, H. Motahari-Nezhad, and F. Toumani. Developing adapters for web services integration. In *Proc. of the 17th International Conference on Advanced Information System Engineering, CAiSE*, pages 415–429. Springer Verlag, June 2005.

[3] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are two web services compatible? In *Proc. 5th International on Technologies for E-Services (LNCS)*, pages 15–28, Toronto, Canada, August 2004. Springer Verlag.

[4] E.-M. Clarke, O. Grumberg, and D.-A. Peled. *Model Checking*, chapter Equivalences and Preorders between Structures, pages 171–178. The MIT Press, Cambridge (Mass.), London, 2001.

[5] R. Djikman and M. Dumas. Service-oriented design: A multi-viewpoint approch. *International Journal of Cooperative Information Systems*, 13(4):337–368, 2004.

[6] M. Dumas, M. Spork, and K. Wang. Adapt or perish: Algebra and visual notation for service interface adaptation. In *Proc. of the 4th International Conference on Business Process Management (BPM)*, pages 65–80, Vienna, Austria, September 2006. Springer Verlag.

[7] M.-C. Fauvet and A. Ait-Bachir. An automaton-based approach for web service mediation. In *Proc. of the International Conference on Concurrent Engineering*, pages 47–54, Antibes, France, September 2006.

[8] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 152–161, Montreal, Canada, October 2003. IEEE Computer Society Press.

[9] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proc. of the 6th International Conference on Enterprise Information Systems*, volume 4, pages 287–295, Porto, Portugal, April 2004. ICEIS Press.

[10] A. Martens, S. Moser, A. Gerhardt, and K. Funk. Analyzing compatibility of bpel processes. In *Proc. of the Advanced Interntional Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)*, pages 147–156, Guadeloupe, French Caribbean, February 2006. IEEE.

[11] P. Oaks and A. ter Hofstede. Guided interaction: A mechanism to enable ad hoc service interaction. *BPM Center Report BPM-05-12*, 2005.

[12] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[13] S. R. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proc. of the International Middleware Conference on Middleware 2004, 5th ACM/IFIP/USENIX*, volume 3231 of *LNCS*, pages 331–351, Toronto, Canada, October 2004. Springer-Verlag.

[14] H. W. Schmidt and R. H. Reussner. Generating adapters for concurrent component protocol synchronisation. In *Proc. of the IFIP TC6/WG6.1 ,The Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, volume 81, pages 213–229, Enschede, The Netherlands, March 2002. Springer-Verlag, Berlin.

[15] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.