Notions of Behavioral Compatibility and their Implications for BPEL Processes

Remco Dijkman

University of Twente, Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE Enschede, The Netherlands
R.M.Dijkman@utwente.nl

Abstract. Informally, two BPEL processes are compatible if they can interact 'successfully'. Different notions exist that we can use to check compatibility. However, each of these notions has a different idea about when interaction is 'successful' and, hence, when two processes are compatible. In this paper we analyze the different notions and define in a precise and unified manner when two processes are compatible according to each of these notions. We also describe the incompatibilities that each notion can and cannot detect. This paper serves three purposes: (i) it can be used as a comprehensive overview of the different notions of service compatibility; (ii) it can be used by designers to select a suitable notion of compatibility; and (iii) it identifies open questions in the area of service compatibility.

1 Introduction

A design question that can be asked when composing web services is whether the services can interact 'successfully'. If they can, we say that they are *compatible*. Although various notions exist to check compatibility, a detailed overview of the incompatibilities that each of them can and cannot detect is lacking.

Therefore, the goal of this paper is to: (i) collect the notions of compatibility that exist and present them in a unified manner; (ii) precisely analyze which incompatibilities each notion can and cannot detect; and (iii) identify open issues in the area of service compatibility. We focus on compatibility of web services that are composed using the Business Process Execution Language (BPEL) [3].

This paper is structured as follows. Section 2 presents the formal model on which we define the notions of compatibility. It also explains the relation between the formal model and BPEL. Section 3 presents the notions of compatibility and section 4 discusses the incompatibilities that each of the notions can and cannot detect. Section 5 presents existing algorithms to check compatibility according to the notions from section 2. Section 6 presents our conclusions.

2 A Formal Behavior Model for BPEL Processes

This section describes the formal model that we developed for Service-Oriented Design [5] and relates BPEL processes to this formal basis.

2.1 Formal Behavior Model

Our formal model for Service-Oriented Design is based on Petri nets with which we assume the reader is familiar. Here, we present a version that is adapted for BPEL. [5] presents a more detailed and general version. In this paper we use a particular form of Petri nets called *workflow nets* [2]. It has been shown by others [11,16] that workflow nets are suitable for representing BPEL processes.

We represent the behavior of a service, or some interacting services, by a labeled Petri net. A labeled Petri net is a tuple (P, T, F, l), such that:

- P is the set of places, which is partitioned into P^I that represents the set of places that are internal to some service and P^C that represents the (connector) places on which (tokens representing) messages exchanged between services are stored.
- T is the set of transitions, which is partitioned into T^I that represents the set of transition that are internal to some service and T^S and T^R that represent the set of transitions that correspond to sending and receiving messages, respectively.
- $F \subseteq (T^S \times P^C) \cup (P^C \times T^R) \cup (T \times P^I) \cup (P^I \times T)$ is the flow relation that connects internal places to transitions and connector places to send and receive transitions. Every connector place must be connected to exactly one send and one receive transition, because connector places are only used to connect send transitions to receive transitions.
- $l : (T^S \cup T^R \rightarrow Msg) \cup (T^I \rightarrow \{\tau\})$ is the labeling function that labels send and receive transitions with the message sent or received and internal transitions with the 'silent' label τ that represents that nothing that is externally observable happens. Send and receive transitions that are connected to the same connector place by the flow relation must have the same label. The label represents the message exchanged. In the paper we sometimes write a '!' or a '?' behind the label of a send or receive transition, respectively. This is *not* part of the label itself.

$M : P \rightarrow \mathbb{N}$ represents the marking of a Petri net.

Workflow nets satisfy the criteria that:

1. an 'initial' place $i \in P$ and a 'final' place $f \in P$ exist, such that i does not have incoming flows and f does not have outgoing flows;
2. all places and transitions are on a path from i to f; i.e.: if we add a transition t that connects f to i by the flow relation (i.e. $\{(f, t), (t, i)\} \subseteq F$), then, for every two places or transitions x and y, there is an (indirect) flow from x to y and vice versa.

In a workflow net $M^i = \{(i,1)\}$, the *initial marking* that has only one token on i, marks the beginning of a process and $M^f = \{(f,1)\}$, the *final marking* that has only one token on f, marks the completion of a process.

We denote possible firing of transition t in a Petri net N with marking M as: (N, M) $[t\rangle$. We denote firing of transition t in a Petri net N with marking M, causing it to change into a marking M' as: (N, M) $[t\rangle$ (N, M'). We denote a sequence of transitions

as t^*; if all transitions labeled τ, we write τ*. We denote the successive firing of a sequence of transitions t^*, causing a Petri net N with marking M to change into a marking M' as: (N, M) $[t^*\!>$ (N, M'). We say that a marking M' is *reachable* from a marking M in a net N, if there exists a sequence of transitions t^*, for which (N, M) $[t^*\!>$ (N, M'). The set of reachable markings from a marking M in a net N is the set that contains all markings M' for which there exists a t^* such that (N, M) $[t^*\!>$ (N, M'). We denote the set of reachable markings as: $(N, M)[\,>$. For a precise definition of firing and reachability, we refer to the Petri net theory [13].

In this paper we refer to the elements of a net by subscripting the element with the name of the net. For example, we refer to the places P of a net N as P_N.

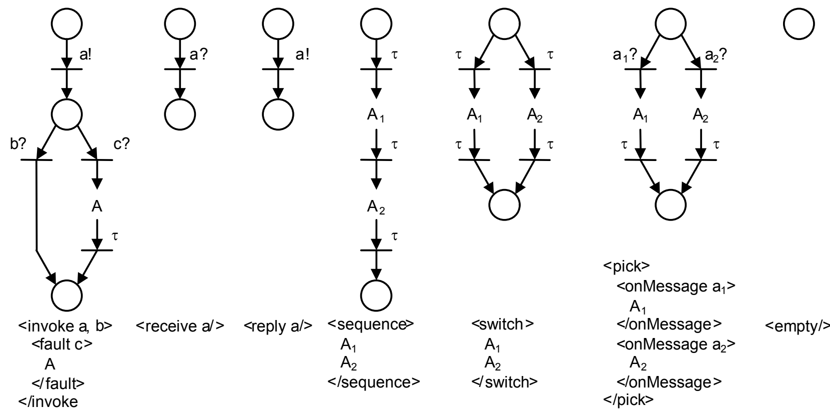


Fig. 1. Transformation of a BPEL Process to the Formal Behavioral Model

2.2 BPEL Process

Figure 1 shows how to transform BPEL activities to the formal behavior model. The shown transformation is incomplete. We only discuss the BPEL activities that we use later on to show which incompatibilities can or cannot be detected. For a more complete transformation we refer to [16].

During an 'invoke' a request message (named *a* in the figure) is sent. In case of a request/response invocation the activity then waits for a response message (named *b* in the figure) or a fault message (named *c* in the figure). After a fault message, the activity contained in the fault is performed. The response and the faults can be left out if desired. During a 'receive', a request message can be received. During a 'reply' a response or a fault message is sent. A 'sequence' performs the contained activities in sequence. A 'switch' performs the contained activity for which the specified condition is satisfied. We do not consider switch conditions here. A 'pick' waits for a message to arrive and then performs the activity contained in the corresponding 'onmessage' statement. During an 'empty' nothing happens.

A complete BPEL process is defined as a tree, in which activities can contain sub-activities. It can be transformed as follows. We transform the root activity according to the transformations from figure 1. For an activity with sub-activities, we put the transformations of those sub-activities in the designated place. Process *Q* from figure 2 illustrates this for a sequence that contains a receive and an empty activity.

Table 1. Different Notions of Compatibility

<table>
<tr><th></th><th>Synchronous</th><th>Asynchronous</th></tr>
<tr><td>Completion</td><td>Synchronous Completion Compatibility</td><td>Asynchronous Completion Compatibility</td></tr>
<tr><td>All Receptions</td><td rowspan="3">Synchronous Simulation Based Compatibility</td><td rowspan="3">Asynchronous Simulation Based Compatibility</td></tr>
<tr><td>Simulation</td></tr>
<tr><td>Bi-Simulation</td></tr>
<tr><td>Refinement</td><td>Synchronous Refinement Compatibility</td><td>Asynchronous Refinement Compatibility</td></tr>
</table>

3 Notions of Behavioral Compatibility

This section discusses three notions of compatibility in this section: *completion, simulation* and *refinement compatibility*. These notions can be checked using compatibility based on *synchronous* and based on *asynchronous* communication. We derived these notions by classifying existing approaches to check compatibility presented in [4,14,15,10,11,12,6,7,5,1]. Section 5 relates these approaches to our classification.

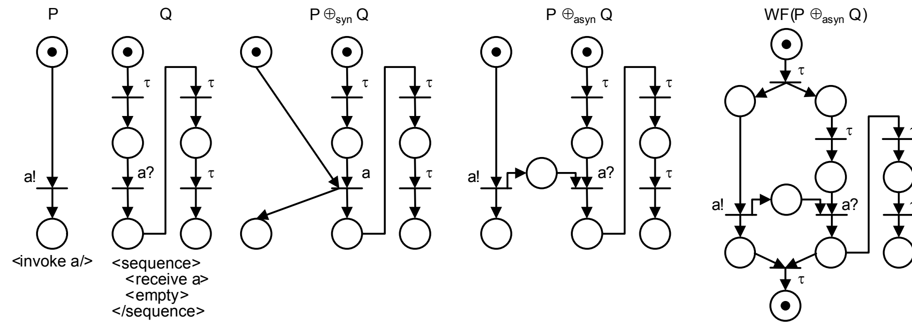


Fig. 2. BPEL Processes and Composition

3.1 Synchronous versus Asynchronous Compatibility

The difference between the synchronous and asynchronous communication model is as follows. In the synchronous model a send and a receive transitions must occur at the same time, while in the asynchronous model a receive transition can occur at any time after a send transition has occurred.

Figure 2 illustrates the difference, by composing two BPEL processes, P and Q, using both asynchronous ($\oplus_{asyn}$) and synchronous ($\oplus_{syn}$) communication. When composing P and Q using asynchronous communication, a connector place is put in between send and receive transactions. Using synchronous communication each pair of a send and a receive transition is transformed into a single transition.

3.2 Completion Compatibility

The notion that we call *completion compatibility* states that two BPEL processes are compatible if and only if both processes can always complete when they are composed. In this paper we focus on asynchronous completion compatibility.

More precisely, we can asynchronously compose Petri net N, with initial marking M_N^i and final marking M_N^f, and Petri net K, with initial marking M_K^i and final marking M_K^f, provided that they have disjoint sets of places and disjoint sets of transitions. These nets have the set of interactions I, which is the set that couples send events from one net to receive events with the same label from the other net:

$I = \{(t^s, t^r) | t^s \in T_N^S, t^r \in T_K^R, l_N(t^s) = l_K(t^r)\} \cup \{(t^s, t^r) | t^s \in T_K^S, t^r \in T_N^R, l_K(t^s) = l_N(t^r)\}$

The asynchronous composition of the nets ($N \oplus_{asyn} K$) is the Petri net (P, T, F, l), for which:

$P = P_N \cup P_K \cup P^C$, where P^C is a set of new places, one for each interaction, such that $P_N \cap P^C = \emptyset$, $P_K \cap P^C = \emptyset$ and $|I| = |P^C|$. Because there exists exactly one place for each interaction, there exists a bijective function $id : I \to P^C$.

$T = T_N \cup T_K$

$F = F_N \cup F_K \cup \{(t^s, p^c) | (t^s, t^r) \in I, p^c = id((t^s, t^r))\}$
$\cup \{(p^c, t^r) | (t^s, t^r) \in I, p^c = id((t^s, t^r))\}$

$l = l_N \cup l_K$

This Petri net has initial marking $M_N^i \cup M_K^i$ and final marking $M_N^f \cup M_K^f$.

A Petri net N is said to *complete*, if and only if the final marking is reachable from the initial marking and from any marking that is reachable from the initial marking:

1. $M^f \in (N, M^i)[>$; and
2. if $M \in (N, M^i)[>$ then $M^f \in (N, M)[>$.

To ensure that we do not falsely conclude that a process is completed, because the final place is marked while there are still active transitions, we add the constraint that the final place can only be marked if the process is completed. Formally, for a net N:

3. for all markings $M \in (N, M^i)[>$: if for some $n > 0$: $(f, n) \in M$, then $M = M^f$.

The resulting net is not a workflow net, because it has more than one initial and final place. However, we can easily transform the net into a workflow net. To do this we add a single initial place with a transition that puts tokens on the initial places of the original nets. We add a single final place on which a token is put by a transition that is enabled if there are tokens on all the final places of the original nets. Figure 2 illustrates this transformation, which we call *WF* in the figure.

3.3 Simulation Based Compatibility

We consider three compatibility notions that are based on simulation: simulation compatibility, all receptions compatibility and bi-simulation compatibility.

The notion of *simulation compatibility* is useful if we distinguish clients from servers. A BPEL process spawns new instances of itself on behalf of its client. We then call it a server with respect to the interactions that it has with its client. However, it may be a client with respect to other interactions.

A client is *synchronous* simulation compatible with a server, if, whenever the client can send or receive a message, the server can perform the opposite action (receive or send the same message). The client and the server can perform internal actions (transitions labeled τ) independent of each other.

More precisely, a Petri net K synchronously simulates a Petri net N, if and only if there exists a relation R that relates states (i.e. markings) in N to states in K, in which K can take transitions with the all the same labels as the transitions that N can take:

1. $(M_N{}^i, M_K{}^i) \in R$; and
2. if $(M_N, M_K) \in R$, and for some $t \in T_N{}^I$: $(N, M_N)[t>(N, M_N')$, then $(M_N', M_K) \in R$
3. if $(M_N, M_K) \in R$, and for some $t^s \in T_N{}^S$: $(N, M_N)[t^s>(N, M_N')$, then there exists some opposite action $t^r \in T_K{}^R$, $l_N(t^s) = l_K(t^r)$ such that there exists a path: $(K, M_K)[\tau^*>(K, M_K{}^1)$, $(K, M_K{}^1)[t^r>(K, M_K{}^2)$, $(K, M_K{}^2)$ $[\tau^*>(K, M_K')$ for which $(M_N', M_K') \in R$.
4. if $(M_N, M_K) \in R$, and for some $t^r \in T_N{}^R$: $(N, M_N)[t^r>(N, M_N')$, then there exists some opposite action $t^s \in T_K{}^S$, $l_K(t^s) = l_K(t^r)$ such that there exists a path: $(K, M_K)[\tau^*>(K, M_K{}^1)$, $(K, M_K{}^1)[t^s>(K, M_K{}^2)$, $(K, M_K{}^2)$ $[\tau^*>(K, M_K')$ for which $(M_N', M_K') \in R$.

We need to adapt the notion of simulation compatibility to consider that a process can be a server with respect to some interactions and a client with respect to other interactions. We can achieve this by parameterizing the check with a set of labels $I \subseteq L$ for which K must simulate N. Then, before performing the check, we change all transitions with labels outside of I into internal transitions labeled τ. Since, client and server processes perform internal transitions independent of each other, this means that transitions with labels outside of I are ignored when checking compatibility.

A process is *synchronous bi-simulation compatible* with another process, if, whenever one partner can send or receive a message, the other partner can perform the opposite action. Partners can perform internal actions (transitions labeled τ) independent of each other. This notion requires both partners to perform the opposite behavior of each other, not just the server to perform the opposite behavior of the client. Hence, it is formalized by a *symmetric* relation that satisfies the rules above.

A process is *synchronous compatible with all receptions*, if, whenever one partner can send a message, the other partner can receive that message. This implies that, one partner can be ready to receive a message, even though the other never sends that message. Partners can perform internal actions (transitions labeled τ) independent of each other. This form of compatibility is formalized by a symmetric relation that satisfies the first three rules above (so it does not need to satisfy rule 4, which states that whenever one partner can receive a message the other must send that message). We refer to [9] for precise explanation of the notions of simulation.

A client process is *asynchronous* simulation compatible with a server process, if, whenever the client can send or receive a message, the server can perform the opposite action *in some possible future (marking)*. The definition of the asynchronous notions using our formalism is left for future work.

3.4 Refinement Compatibility

Using the notion of refinement compatibility, a designer *first* specifies what it wants processes to do in their composition. We call this specification a *choreography*. A process is compatible with other processes, if it has the same behavior as the part of the choreography that it performs, after abstracting from the part of its behavior that does not relate to the choreography. We can check compatibility of a process *P* that performs a part of a choreography *C* as follows [5]:

1. Remove the part of *C* that *P* does not perform, resulting in *C*';
2. Abstract from the part of *P* that does not relate to *C*, resulting in *P*' (we abstract from a transition by labeling it τ);
3. Check if *C*' is (bi-simulation) equivalent to *P*'.

For example, consider $WF(P \oplus_{asyn} Q)$ from figure 2 a choreography that represents the joint behavior of two parts *P* and *Q*. Using refinement compatibility we can check if *E* from figure 3 correctly fulfills the behavior of *Q*. This is the case, because if we abstract from *b*?, which is outside the choreography, then *E*' is bi-simulation equivalent to *Q*.

Due to space limitations, we restrict ourselves to a brief informal explanation of refinement compatibility. For a precise definition, we refer to [5].

Table 2. Incompatibilities and Their Detection by Different Notions of Compatibility

Y = incompatibility can be detected by notion N = incompatibility can not be detected by notion (false positive) F = incompatibility can falsely be detected by notion (false negative)		Synchronous					Asynchronous				
		Completion	Simulation	All Receptions	Bi-simulation	Refinement	Completion	Simulation	All Receptions	Bi-simulation	Refinement
completion incompatibility	branching incompatibility	Y	N	Y F	Y F	Y	Y	N	Y F	Y F	Y
	order incompatibility	N F	N F	N F	N F	N F	Y	Y	Y	Y	Y
	server non-completion incompatibility	Y	Y / N[3]	N	Y	Y	Y	Y / N[3]	N	Y	Y
goal incompatibility	unreachable goal	N	Y F[1]	N F	Y F	Y	N	Y F[1]	Y F	N F	Y
	asymmetric goal reachability	N	N	N	N	Y	N	N	N	N	Y
refinement incompatibility	interface incompatibility	N	Y F[1]	Y F[2]	Y F	Y	N	Y F[1]	Y F[2]	Y F	Y
	policy incompatibility	N	N	N	N	Y	N	N	N	N	Y
incompatibility other aspect		N	N	N	N	N	N	N	N	N	N

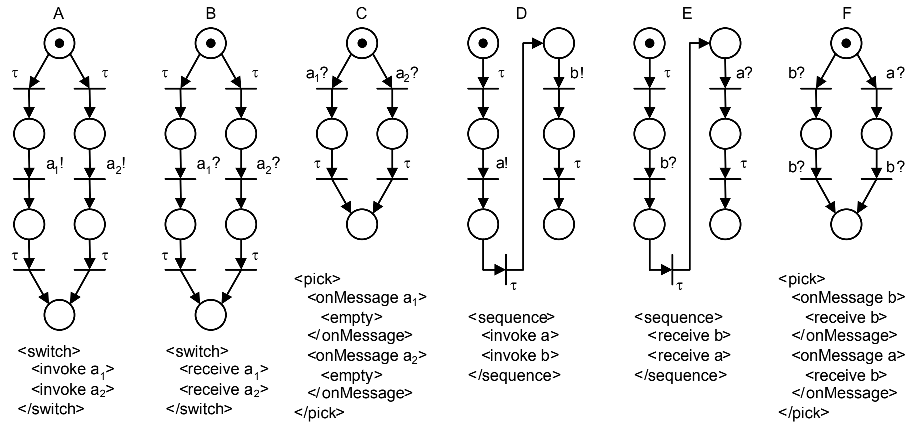
1 Detected only on client-side
2 Detected only with respect to receptions
3 Depending on whether ready simulation is used or not

4 Incompatibilities Detected by and Limitations of the Notions

Although the notions explained in section 3 are called compatibility notions, they do not prove *compatibility* of web services. Because, even if two web services are compatible according to a notion, they may be considered incompatible by the designer. Rather the compatibility notions prove *incompatibility* of web services.

For example, if two web services are ‘compatible’ according to the completion compatibility notion, they can still be incompatible in the sense that they deadlock because one path in a ‘switch’ can never be taken (the path in a switch that is taken depends on the value of the switch variable, which is not considered in the completion compatibility notion). However, if two web services are incompatible according to the completion compatibility notion, they deadlock and, therefore, are indeed incompatible.

In this section we explain incompatibilities in BPEL processes that can be detected by the notions from section 3. Table 2 displays the incompatibilities that we consider. We derived the incompatibilities by analyzing BPEL processes and deciding which problems can occur when composing them. Although there can exist more incompatibilities than those covered in this paper, we claim that at least the ones listed in the table are relevant. In this section we also show which incompatibilities can and cannot be detected by the various notions, as summarized in table 2.




```
<switch>
  <invoke a1>
  <invoke a2>
</switch>
```

```
<switch>
  <receive a1>
  <receive a2>
</switch>
```

```
<pick>
  <onMessage a1>
    <empty>
  </onMessage>
  <onMessage a2>
    <empty>
  </onMessage>
</pick>
```

```
<sequence>
  <invoke a>
  <invoke b>
</sequence>
```

```
<sequence>
  <receive b>
  <receive a>
</sequence>
```

```
<pick>
  <onMessage b>
    <receive b>
  </onMessage>
  <onMessage a>
    <receive b>
  </onMessage>
</pick>
```

Fig. 3. Compatibilities and Incompatibilities in Detected by Completion Compatibility

4.1 Completion Incompatibilities

Two BPEL processes are incompatible if, in their composition, an execution is possible in which one or both processes do not complete. Because, if a process does not complete it remains active indefinitely consuming resources. Moreover, each new instance of the process may not complete, eventually causing the system to run out of resources and crash. We distinguish three special cases of completion incompatibility.

A *branching incompatibility* exists when processes can make incompatible internal choices, causing one or both of the processes to deadlock. Processes *A* and *B* from figure 3 illustrate this incompatibility. In case *A* takes the 'invoke a_1' case, while *B* takes the 'receive a_2' path, *B* cannot complete. A *server non-completion* incompatibility exists if an execution is possible in which the client completes while the server does not. An *order incompatibility* exists when a process deadlocks in case it receives messages in a different order than the order in which they were sent. Processes *D* and *F* from figure 3 illustrate this. *F* can receive the messages sent by *D* in a different order, causing *F* to deadlock after the sequence *a*!, *b*!, *b*?.

4.2 Goal Incompatibilities

Two processes are incompatible if, in their composition, one process or both processes cannot reach the goal for which the processes are in effect. We call this form of incompatibility *unreachable goal* incompatibility. An example of an unreachable goal incompatibility is a composition in which interaction between two processes always leads to the rejection of an offer. We derived this incompatibility from the definition of a business process as: "*A set of one or more linked procedures or activities which collectively realise a business objective or policy goal …*" [17]. In this definition, realizing a goal is an important property of a process.

Another goal-related incompatibility that can occur is that one process in a composition can conclude that the goal was reached, while the other concludes that it was not. We call this form of incompatibility *asymmetric goal reachability*.

4.3 Refinement Incompatibilities

Two processes are *refinement incompatible* if, in their composition, they do not meet some requirement (implicitly) specified at some higher level of abstraction. In this paper we focus on the following two forms of refinement incompatibility.

An *interface incompatibility* exists when interactions that are required to occur between specified partners at specified interfaces, occur between other partners or at other interfaces. For example, a requirement can be that the 'sale' interaction occurs between a 'buyer' and a 'seller' process. In that case, the composition cannot specify that the 'sale' interaction occurs between the 'seller' and the 'shipper' process.

A *policy incompatibility* exists when a specified (behavioral) policy is not met by a process. An example of a behavioral policy is that a broker must obtain at least three offers before notifying the client of the best offer.

4.4 Incompatibility with respect to aspects other than behavior

The notions of compatibility that we found so far in literature focus on incompatibility with respect to the behavioral aspect. However, other aspects can cause incompatibilities as well. These include:

– timing aspects; for example, a time-out incompatibility exists if one process times-out after some period *t*, while the partner process always delays for more than *t*.
– information aspects; for example, an information incompatibility can occur if a path in a 'switch' cannot be taken (causing a deadlock), because the information value that causes the path to be taken cannot be established.
– addressing aspects; for example, an addressing incompatibility can occur if in one iteration of a 'while' a process interacts with one process, while in the next iteration it interacts with another process (that is in a state in which it is not ready to interact and, therefore, causes a deadlock).

4.5 Incompatibilities detected by the notions

Each of the notions from section 3 can detect some incompatibilities, while it cannot detect others. When a notion cannot detect an incompatibility, this leads to a *false positive* (concluding that two processes are compatible while they are not). In addition to that a notion of compatibility may indicate that there exists an incompatibility where there is none, we call this a *false negative*. Table 2 indicates which incompatibilities can be detected by which notions, as well as false positives and false negatives.

From the table we conclude that the asynchronous refinement notion detects the most of the incompatibilities that we identified. However, we warn the reader that the refinement notion uses a top-down approach, requiring the designer to first specify a required composite behavior in detail. Hence, if the designer rather uses a bottom-up approach, the refinement notion is not suitable.

The completion compatibility notion is especially devised to check completion and can therefore detect all completion incompatibilities. However, the simulation notion has problems detecting non-completion incompatibilities, while the bi-simulation notion and the 'all receptions' notion detect a false branching incompatibility. Processes *A* and *C* from figure 3 illustrate this problem. The two processes are compatible; *A* either takes the 'invoke a_1' or 'invoke a_2' case, upon which *C* either receives a_1 or a_2 and continues along the corresponding path in the 'pick' activity. However, the processes are incompatible according to the bi-simulation notion, because, if they were compatible:
$(M_A^i, M_B^i) \in R$ (*according to rule* 1)
$(A, M_A^i)[\tau\rangle(A, M_A')$ for which $(A, M_A')[a_1!\rangle$, requires $(M_A', M_B^i) \in R$ (*rule* 2)
$(M_A', M_B^i) \in R$ requires $(M_B^i, M_A') \in R$ (*symmetry of R*)
$(B, M_B^i)\ [a_2?\rangle(B, M_B')$ requires $(A, M_A')[a_2!\rangle$ (*rule 4*)
This leads to a contradiction, because in (A, M_A') only $a_1!$ is possible.

Completion of the server process is not a requirement for the simulation and the 'all receptions' notion. Hence, these notions cannot detect a *server non-completion* incompatibility. However, a more refined notion of simulation, called ready simulation [9], can be used that also detects server non-completion.

Synchronous compatibility notions cannot detect all order incompatibilities. Also, synchronous compatibility notions may falsely decide that there is no order incompatibility. These false positives and false negatives are returned by synchronous compatibility notions, because they use a synchronous communication model to check

properties in Web Service communication, which is asynchronous. Processes *D* and *E* from figure 3 illustrate a false negative. In the synchronous model the two processes deadlock. However, the processes can successfully communicate asynchronously through the sequence a!, b!, b?, a?.

In spite of their problems, synchronous compatibility notions are often used, because asynchronous notions may be undecidable in some cases [4,8]. To mitigate the problems with the synchronous notions, [8] describes an approach to check if the interaction between two processes can be analyzed without problems using a synchronous notion.

The refinement notion can detect all incompatibilities with respect to goal reachability, because this notion requires a process to be a correct refinement of a more abstract process or choreography. Hence, if a goal is reachable in a more abstract process, it must be reachable in the refined process. An unreachable goal incompatibility cannot occur on the client side in simulation compatibility, because this notion requires that each execution of the client process is an execution of the server process. Similarly, this incompatibility cannot occur in bi-simulation compatibility. It can occur in ‘all receptions’ compatibility, if the goal is achieved in the path of a reception that does not occur. Although unreachable goals cannot exist in processes that are simulation or bi-simulation compatible, these notions cannot be said to ‘detect’ goal unreachability. They merely ensure that *all* executions occur and therefore also the execution in which the goal is reached. Therefore, these notions also detect false negatives which respect to goal reachability, because they may reject a composition in which some non-essential execution (i.e. an execution that is not required to reach the goal) is not possible. For example, a client that is interested in buying products of two different types, but will buy if only one type is available, is compatible with a server that only sells one type. However, the simulation notions will falsely result in an incompatible verdict.

Refinement incompatibilities are only detected by the refinement notion. Refinement notions can detect these incompatibilities, because they allow the designer to specify requirements, such as policy or interface requirements, in the choreography and then check if the processes satisfy the choreography.

None of the notions we investigated so far can detect incompatibilities with respect to aspects other than behavior.

Table 3. Existing Approaches to Compatibility Checking

	Synchronous					Asynchronous				
	Completion	Simulation	All Receptions	Bi-simulation	Refinement	Completion	Simulation	All Receptions	Bi-simulation	Refinement
Bordeaux et al. [4]	X		X	X						
Massuthe et al. [14,15]							X[1]			
Martens [10,11,12]						X				
Foster et al. [6,7]										X
Dijkman and Dumas [5]										X
van der Aalst and Weske [1]										X

1 Uses a form of ready simulation. Hence, it can detect server non-completion.

5 Existing Approaches to Check Compatibility

Table 2 shows which compatibility notions are addressed by existing approaches to check compatibility.

Bordeaux et al. discuss three notions of compatibility. Their work is limited to compatibility between two interacting processes and externally observable behavior (i.e. silent transitions are not allowed). They do not discuss algorithmic approaches. Massuthe et al. present an algorithmic approach to check asynchronous (ready) simulation compatibility. At the moment their approach is restricted to processes without silent transitions and without loops. However, research is ongoing.

Martens presents an algorithmic approach to check asynchronous completion compatibility. His work specifically targets compatibility of BPEL processes.

Foster et al., Dijkman and Dumas, and van der Aalst and Weske present approaches to check asynchronous refinement compatibility. The work by Dijkman and Dumas and by van der Aalst and Weske closely resembles the notion discussed in section 3.4. The work by Foster et al. differs from theirs, because process requirements can be specified separately (e.g. in separate message sequence diagrams) instead of in a single choreography. Foster et al. can also deal with information aspects in a simplified form.

6 Conclusion

This paper presents an overview of the different notions of behavioral compatibility between web services. It defines these notions on a Petri net formalism and precisely analyses which incompatibilities each of the notions can and cannot detect.

We conclude that notions exist to check compatibility with respect to completion (can web services always complete if they are composed?). Also, a notion exists to check compatibility with respect to goal reachability (can a composition of web services reach a specified goal?).

The drawback of the 'refinement' notion, which is the only notion that can check all types of goal reachability, is that it requires a designer to specify the required behavior of services in their composition. This requires significant effort, which the designer may not want to spend. Hence, it motivates the development of a more lightweight approach that allows us to check goal reachability.

Compatibility with respect to 'interface structure' (does the partner process support the port types required or provided?) and 'policies' (does a process meet the policies specified?) can only be checked by 'refinement' notions. Compatibility with respect to other aspects such as information, time and address cannot be checked by any of the notions. Hence, development of approaches that can check compatibility with respect to these aspects is still needed.

Acknowledgements

The author thanks Dick Quartel for his comments on an earlier version of this paper. He also thanks Marlon Dumas for providing useful references about service compatibility. The work described in this paper is partly supported by the European Commission through the SPICE project under contract IST-027617.

References

1. van der Aalst, W., Weske, M.: The P2P approach to Interorganizational Workflows. In: Proc. of the Int. Conf. on Advanced Information Systems Engineering. Lecture Notes in Computer Science, Vol. 2068. Springer-Verlag (2001) 140-156
2. van der Aalst, W.: Verification of Workflow Nets. In: Application and Theory of Petri Nets. Lecture Notes in Computer Science, Vol. 1248. Springer-Verlag (1997) 407-426
3. Andrews, T., et al.: Business Process Execution Language for Web Services Version 1.1. OASIS Specification (2003)
4. Bordeaux, L., Salaün, G., Berardi, D., Mecella, M.: When are Two Web Services Compatible? In: Int. Workshop on Technologies for E-Services. Lecture Notes in Computer Science, Vol. 3324. Springer-Verlag (2005) 15–28
5. Dijkman, R., Dumas, M.: Service-Oriented Design: A Multi-Viewpoint Approach. Int. J. of Cooperative Information Systems 13 (2004) 337-368
6. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Compatibility Verification for Web Service Choreography. In: Proc. of the Int. Conf. on Web Services (2004)
7. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Compositions. In: Proc. of the Int. Conf. on Automated Software Engineering (2003)
8. Fu, X., Bultan, T., Su, J.: Analysis of Interacting BPEL Processes. In: Proc. of the Int. World Wide Web Conf. (2004) 621-630
9. van Glabbeek, R.: The Linear Time – Branching Time Spectrum I: The Semantics of Concrete Sequential Processes. In: Handbook of Process Algebra. Elsevier (2001) 3-99
10. Martens, A.: Analyzing Web Service based Business Processes. In: Proc. of Int. Conf. on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, Vol. 3442. Springer-Verlag (2005)
11. Martens, A.: Simulation and Equivalence of BPEL Process Models. In: Proc. of the Design, Analysis, and Simulation of Distributed Systems Symposium (2005)
12. Martens, A: On Compatibility of Web Services. Petri Net Newsletter 65 (2003) 12-20
13. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proc. of the IEEE 77 (1989) 541-580
14. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics, 1 (3) (2005) 35-43
15. Massuthe, P., Schmidt, K.: Operating Guidelines – an Automata-Theoretic Foundation for the Service-Oriented Architecture. In: Proceedings of the IEEE Int. Conf. on Quality Software (2005) 452-457
16. Verbeek, H., van der Aalst, W.: Analysing BPEL Processes Using Petri Nets. In: Proc. of the Int. Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management (2005) 59-78
17. Workflow Management Coalition: Workflow Management Coalition Terminology and Glossary. Technical Report WFMC-TC-1011 (1999)