

# Algorithms Based on Pattern Analysis for Verification and Adapter Creation for Business Process Composition

Akhil Kumar and Zhe Shan

Department of Supply Chain and Information Systems, Smeal College of Business,  
Penn State University, University Park, PA 16802, USA  
{akhilkumar,zheshan}@psu.edu

**Abstract.** With more automation in inter-organizational supply chains and proliferation of Web services technology, the need for organizations to link their business services and processes is becoming increasingly important. Ideally, such linking must be automated and also possible to do on-the-fly in an ad hoc manner. In this paper, we view business processes in terms of standard patterns, and describe a pattern compatibility matrix and rules that allow us to simplify the task of checking compatibility between two or more processes because these prerequisite rules can be applied to each pattern separately, thus reducing the search space. We give an algorithm for applying these rules to check process compatibility. If two processes are compatible, we determine whether an adapter is required, and if so, a minimal adapter is generated by another algorithm. Two variants of the algorithm (PBA and PBA-MIN) are implemented, and experimental results and comparisons with an existing algorithm are given.

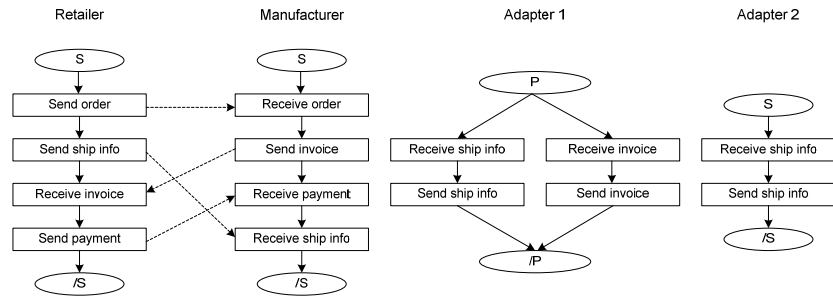
**Keywords:** Web services, verification, composition, adapter generation, pattern compatibility matrix, minimal adapter, business process complexity metrics

## 1 Introduction

Web services are loosely coupled, distributed reusable components that encapsulate discrete functionality and are accessible over standard XML-based Internet protocols such as SOAP and WSDL [23]. The grand vision is of lots of web services leading to considerable machine-to-machine interaction, automated, "on-the-fly" composition of two or more services to create new services, and new technologies for designing and offering services. This is a natural progression of the Web from static (information only) web sites to dynamic web sites that allow transactions, and further to "intelligent" interactions enriched with semantics.

In general, multiple composite web services may interact with one another in complex patterns. To motivate the need for an adapter, consider for instance Figure 1 that shows snippets from an order handling example involving a retailer and a manufacturer, each running its own process. After an *order* is placed by the retailer and received by the manufacturer, the retailer sends another message containing the

shipping information (*ship info*), while the manufacturer sends the *invoice*. Subsequently, the retailer receives the *invoice* and sends the *payment*. Later, the manufacturer receives the *payment* and then receives the *ship info*. Thus, *ship info* is received after *payment*, even though it is sent earlier in the manufacturer's process. It shows that, assuming synchronous communication and no buffering of messages, these two processes can work together only if an adapter process is added as shown to ensure proper operations. Note that in the absence of an adapter, the processes would get stuck at *send ship info* and *send invoice* and would not be able to proceed. Two possible adapters are shown and will be discussed in more detail later.



**Figure 1:** An example to illustrate the need for an adapter

The example processes of Figure 1 are sequential, but, in general, they may involve parallel, choice, loop and other patterns. Moreover, there are different possible scenarios for how interactions between a service client (the retailer, in this example) and various service providers (such as the manufacturer) may unfold. More complicated scenarios may involve the client interacting with provider 1 which may in turn call provider 2, and provider 2 may return a final answer to the retailer. When web services interact in a meaningful way, the sequence of interactions is called a choreography [4]. Recently, standards and languages for defining such interactions are also beginning to emerge [24,25,29].

In general, even though the exposed interfaces of two processes are well matched, yet at the behavior level they may fail to interoperate because of the interaction patterns. Therefore, in addition to the interface matching, it is necessary to know the actual behavior within each process, and determine if two or more processes are compatible. This requires the development of algorithms to check for compatibility. First, there is a need to check if two processes are deadlock-free, which means that there is no cyclic dependency among the activities of two processes. Moreover, in many situations even if two processes are deadlock-free, a mediator or adapter may still be required to ensure proper operation as in the above example. Therefore, it is necessary to determine if an adapter is required and also to generate an adapter process in an automated manner.

Consequently, in this paper we address the problem of how to develop an algorithm to ensure that independent business processes (or composite Web services) can be composed "correctly". This requires development of verification techniques. We have developed a novel approach based on structural analysis of patterns which we show is

more efficient than existing approaches. The general problem we address is to verify that two processes can interoperate correctly at the behavior level given that the communication interfaces have been well matched and the behavior of each process is correct by itself. Our notion of correctness is proper termination. Methods for checking a single process are well-known [15,28].

In this paper we assume that each process is well structured [15], i.e. each split for a pattern has a corresponding join, and also that the patterns are properly nested inside one another. Moreover, it is assumed that the messages in interacting processes have been well matched, i.e. each send activity in process 1 has a corresponding receive activity in process 2, and vice versa. Additionally, we assume that each send or receive activity is unique and is not repeated in the process. Finally, we consider both synchronous and asynchronous modes of communication between processes. For both synchronous and asynchronous process models, our algorithm can help to verify the compatibility between two processes. If they are compatible, no adapter is needed for the asynchronous model. For the synchronous model, while checking the compatibility, our algorithm can also generate the adapter if necessary.

We implemented an initial prototype system in Java, which reads WSBPEL files as input and outputs diagnoses based on the algorithm result. Also, we built a testing environment for process composition, which includes a random generator of test cases and an execution simulator for process composition. The test result validates the correctness of our algorithms. Furthermore, the performance evaluation shows the advantages of our two algorithms compared to the SET approach in [5].

Consequently, the plan of this paper is as follows. The next two sections describe interaction modes and patterns that arise when two processes interact through a mediator. Section 4 gives our verification and adapter creation algorithms in detail. Then, Section 5 describes our prototype implementation, and results of testing and performance evaluation. Later in Section 6, a detailed discussion of our approach in the context of related work is given. Section 7 concludes the paper.

## **2 Preliminaries - Synchronous and Asynchronous Interaction**

Business processes discussed in this paper illustrate the behavior two interacting services. We can map this specification onto WSBPEL [24] to implement the business processes. Processes interact in two modes: *synchronous* and *asynchronous*. We use WSBPEL to illustrate these notions. In WSBPEL the *invoke* activity may be used by one process (say a client) to call a partner web service (say, a server) either synchronously or asynchronously. In general, in a synchronous web service call, a caller passes both input and output variables and then waits for a reply or "blocks" until a reply is received. On the other hand, in an asynchronous call, only an input variable is passed and the caller resumes processing without waiting for a reply. We assume that the underlying transport mechanism ensures reliable message delivery.

In the asynchronous mode of interaction, a message received by a server is placed in a random access buffer until explicitly requested by the receive activity. A WSBPEL server like Oracle BPEL Process Manager maintains message buffers. Enabling such a buffer, the application server can (i) save the received message in a buffer, (ii) send the acknowledge message back to the sender, and (iii) interact with the receive activity to deliver the message when the receive activity is ready.

On the other hand, when a service is called a client making a synchronous call blocks until a reply is received. Thus, if both the client and the server processes are performing a send, then this causes the processes to get stuck since there is no buffer to store a message and a send must be immediately received. This gives rise to a need for an adapter as we saw in the example of Figure 1. However, under the asynchronous interaction assumption, for the example in Figure 1 we will not need an adapter because both sides can buffer the messages they receive if the corresponding activity is not ready. Thus, the server at the manufacturer will buffer the message *ship info* until the process reaches the *receive ship info* activity and is able to consume the buffered message. Nevertheless, when the processes involve more complicated interactions, the buffer mechanism may also lead to problems. In particular when one party makes a choice, the other party should also make the same choice or else the processes will deadlock. In such a situation, the manufacturer process is not able to proceed without some kind of exceptional handling effort. To some extent this also represents poor process design. The manufacturer should not make a choice between cash and check payment unless it has agreed with the retailer on the form of payment. We treat such cases as examples of poor process design, and exclude them for now.

**Table 1:** Scenarios to which our algorithms apply

Communication model	Verify algorithm	Create adapter algorithm
Synchronous	yes	yes
Asynchronous	yes	n/a

Table 1 shows the main scenarios we consider in this paper. Basically, the verify algorithm works for both of these scenarios, while the adapter creation algorithm is relevant only to the synchronous model because an adapter is not required for the asynchronous model.

### 3 Interaction Patterns

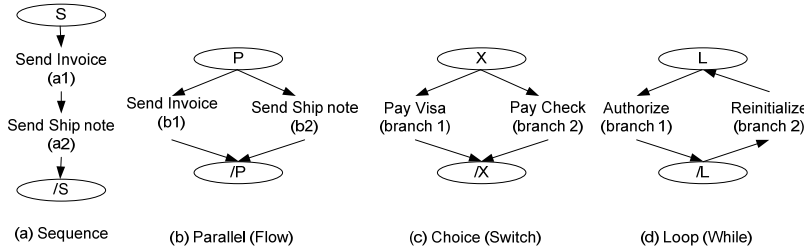
There are two modes of web services interaction: *simple* and *complex*. In the simple mode, a service exposes its interface in terms of operations. A client accesses the operations using the signature of the service described, say in WSDL [23]. The client sends a request packaged in a SOAP envelope, and the service sends a reply again in a SOAP envelope that the client can understand [23]. Thus, the client can avail of the service by using appropriate operations in XML. The sequence of operation calls is not important. Mismatches between signatures of the client and the server can also be resolved through a mediator.

In the complex mode of operation, the service itself can be viewed as a process. Thus, the client may interact with the service through a series of atomic interactions; however, the interaction must be performed in a certain coordinated manner in order to be correct and consistent with the data flow required by the service.

#### 3.1 Complex Process Interaction Patterns and control flow mediation

A web service, say, one described in WSBPEL, is essentially a process, and interactions between web services can also be viewed as process interactions [1,5,20, 22]. A process is composed of basic patterns or building blocks like *sequence (SEQ or*

*S*), *parallel* (*PAR* or *P*), *choice* (*XOR* or *X*) and *loop* (*L*) as shown in Figure 2. These patterns can be combined to create processes in WSBPEL. The parallel pattern contains two (or more) branches that can be interleaved, while the choice pattern contains multiple branches of which only one is selected at run time. Finally, the loop pattern contains one forward branch and one backward branch that may be repeated multiple times. For two interacting loop patterns, their iteration counters should be equal, or their loop conditions should be identical. Each branch may be a single activity, or in general, it may represent a child process.



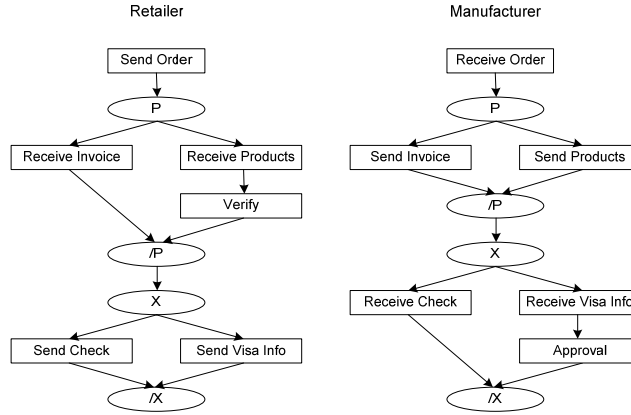
**Figure 2:** Basic process patterns (and their corresponding names used in WSBPEL)

As an example, consider Figure 3 where each process is composed of two basic patterns: parallel and choice which are in turn connected with one another in a sequence. After the products have been ordered by the retailer, the retailer process waits to receive an *invoice* and also the *goods* in any order. Similarly, the manufacturer process may send the *invoice* and *goods* in any order. Upon receiving the *goods* and *invoice*, the retailer may make the payment either by *check* or by *visa*. The manufacturer process will receive the corresponding payment. Also, note that the *Verify* and *Approval* steps are internal steps in each process and they involve no interaction with the other process. They are disregarded in verification.

This figure shows that the two processes can interoperate correctly because: (1) each pattern in the retailer process has a matching pattern in the manufacturer process and vice versa; (2) every send action in a pattern has a corresponding receive action in its corresponding pattern; (3) there is no deadlock.

### 3.2 Pattern compatibility matrix

The *basic patterns* described above, like *sequence*, *choice*, *parallel* and *loop*, can be combined to create *composite patterns*. A pattern consisting of *only sequence and parallel* structures is called an *SP pattern*. When processes interact, clearly one basic requirement for compatibility is that each send in pattern 1 must have a corresponding receive in pattern 2, and vice versa; else the two processes cannot interoperate. However, as noted earlier this rule by itself does not guarantee interoperability. Our goal here is to look for ways to check interoperability at a higher level. Hence, we introduce the notion of pattern compatibility. Table 2 is a pattern compatibility matrix showing compatibility between patterns in different processes. When processes interoperate, the patterns inside them must also interoperate according to the rules in this matrix, i.e. they should have corresponding or matching patterns.



**Figure 3:** An example of interaction between processes involving parallel and choice patterns

**Table 2:** A pattern compatibility matrix

	Sequence	Parallel	Choice	Loop	SP
Sequence	Yes	Yes	No	No	Yes
Parallel	Yes	Yes	No	No	Yes
Choice	No	No	Yes	No	No
Loop	No	No	No	Yes	No
Seq-par	Yes	Yes	No	No	Yes

Below we give a simple result to capture the essence of this matrix.

**Result 1:** The following compatibility rules (see Table 2) must be observed between interacting patterns of different processes:

- 1) An SP pattern from process 1 must match with an SP pattern from process 2.
- 2) An X pattern from process 1 must match with an X pattern from process 2.
- 3) An L pattern from process 1 must match with an L pattern from process 2.

*Proof Sketch.* We argue in terms of action sets, i.e. all the atomic actions in a pattern. An S and P pattern can match each other if they contain the same action set because all actions are executed. X cannot match with S, P or L patterns because only one branch of an X can be executed, and so the action set of the executed branch will not be the same as the action set of the other patterns. Finally L can only match with an L because its action set can be repeated multiple times. Hence, Table 2 is valid. Moreover, this is a necessary condition for process compatibility.

**Example 1:** In Figure 3, the two parallel patterns in the retailer and manufacturer processes match because the *send invoice* and *receive invoice* activities correspond to each other (as do, *send products* and *receive products*). Similarly, the choice patterns also match because their branches have corresponding matching branches.

The next example shows how two SP patterns can interoperate with each other.

**Example 2:** Figure 4 shows process snippets to illustrate interoperation between a sequence and SP patterns. After receiving the retailer's order, the manufacturer sends

the *invoice* and the *ship info* (shipping information) in parallel; however, the retailer can receive them in sequence only as shown. Thus, an adapter must be created. The adapter contains two activities in sequence, *receive ship info* and *send ship info*.

The compatibility rules reduce the search space considerably, and in the next section we will apply these rules to develop an algorithm to verify that two processes can interoperate.

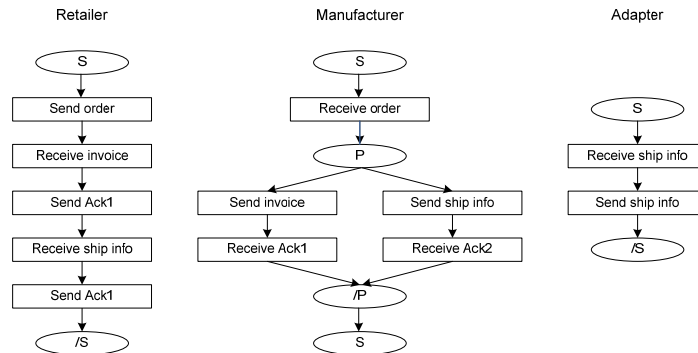


Figure 4: An example to show how S and SP Patterns can interoperate

## 4 Algorithms for Compatibility Verification and Adapter Creation

Now we apply the above rules to design algorithms to first verify if two processes are compatible and then to generate an adapter if it is required. For asynchronous cases, if two processes are verified to be compatible, they can communicate without any adapter. However for synchronous cases, even if two processes are verified to be compatible, they may still need an adapter. Here we first introduce the process normalization algorithm, which is executed on the original processes before they are passed to the compatibility verification and adapter creation algorithms. Then we present the verification algorithm and the adapter creation algorithms. Later, we use two heuristics to improve the SP adapter algorithm to create a minimal adapter.

### 4.1 Process Normalization

The motivation of process normalization is to first make the processes structurally similar for verification. The original processes may contain both private activities, which are internal within a company, and public activities, which are communications with external parties. When checking compatibility, we only need to consider the public activities. Therefore, we first delete all private activities in both processes. After this deletion, the structure of processes may be changed. Hence, we need to reorganize the corresponding patterns according to this procedure:

- 1) Delete private or internal activities.
- 2) If a *non-loop* parent is left with only one child, then simplify by removing parent.
- 3) Merge immediately nested identical, non-loop patterns (e.g. if parent and child are both X)
- 4) If two or more branches of an X pattern are identical, then merge them into one.

## 4.2 Verification Algorithm

To verify whether two processes are compatible, first we need to check whether the two sets of activities can communicate with each other, i.e. for each activity of a process, its dual task is in the other process. After that, we need to check whether there is any deadlock in the communications between two processes. Then, we turn to *choice* and *loop* patterns. We handle these structures by decomposing them. A choice contains multiple branches, each of which can be viewed as a subprocess. Similarly, a loop contains forward and backward branches (or subprocesses) that are repeated 0 or more times depending upon the loop condition. As discussed above, a choice in one process must have a matching choice pattern in the other process, and a loop should likewise have a matching loop pattern. The algorithm for choice and loop is similar. We find all the loop or choice patterns in each process, and then check whether their corresponding branches are compatible too. Therefore, the algorithm will be executed recursively until all loop and choice patterns are verified. Figure 5 gives the full algorithm. This algorithm is called twice, with the arguments P1 and P2 interchanged the second time. If it returns 'True' both times, it means that P1 and P2 are compatible.

```
verify(P1,P2) //verify two processes denoted by P1, P2
{
  foreach (activity t1 in P1)
    if (not exists t2 in P2 s.t. t2 == dual(t1))
      return(False);
  if (deadlock_exists(P1, P2))
    return(False);
  if (P1.type == 'X' || P1.type == 'L')
    if (not exists P2.branch1', P2.branch2' s.t.
        P2.type = P1.type && P1.loop_count=P2.loop_count
        && verify(P1.branch1, P2.branch1')
        && verify(P1.branch2, P2.branch2'))
      return(False)
    else{
      foreach (child X1 in P1)
        if (not exists X2 in P2 s.t. verify(X1,X2))
          return(False)
      foreach (child L1 in P1)
        if (not exists L2 in P2 s.t. verify(L1,L2))
          return(False);
    }
  return(True);
}
```

Figure 5: Verification Algorithm

### Deadlock Checking

As part of the verification algorithm, the deadlock detection algorithm first transforms processes P1 and P2 into a directed graph. In addition to the existing links in P1 and P2, we create links from each send activity to its corresponding receive activity. In this directed graph, we check for cycles, and their absence means P1 and P2 are deadlock free. In general, this can be extended to any number of processes. Figure 6 gives the algorithm details.

```

deadlock_exists(P1, P2)
{
  change loop structures in P1, P2 into sequence;
  transform processes P1, P2 into directed graphs GP1, GP2;
  combine GP1 and GP2 into a new graph GP by adding directed links
    from each send to its corresponding receive activity;
  check for cycles in the graph GP;
  if (any cycle is found) return True;
  else return False;
}

```

**Figure 6:** Deadlock Detection Algorithm

### 4.3 Adapter Creation Algorithms

For synchronous cases, besides verifying that two processes are compatible, we also need to check whether they need an adapter to achieve successful communication. In this section, we introduce the pattern-based algorithm (PBA) for adapter creation. The adapter creation algorithm works bottom up and is discussed next (see Figure 7).

```

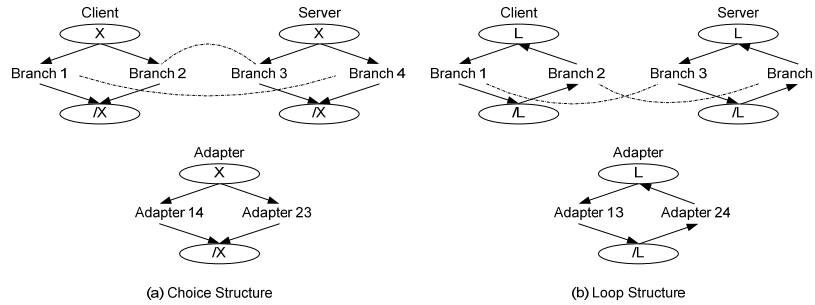
Adapter createAdapter(Proc P1, Proc P2)
{
  //Find innermost matched X/L patterns XL1 and XL2;
  while ((XL1,XL2)!=null
  {
    Adapter={};branchAdapterSet={}; //initialize for pattern adapter
    foreach ((b1,b2)= findMatchedBranchPair(XL1,XL2)) //each branch pair
    {
      branchAdapter = createSPAdapter(b1,b2)
      branchAdapterSet = branchAdapterSet ∪ branchAdapter;
    }
    if (XL1.type == 'X') Adapter = Xor(branchAdapterSet); //merge Xor
    if (XL1.type == 'L') Adapter = Loop(branchAdapterSet); //merge Loop
    blockReplace(P1,XL1,block_new1); //replace XL1 by block_new1 in P1
    blockReplace(P2,XL2,block_new2); //replace XL2 by block_new2 in P2
    blockAdapter[block1][block2] = Adapter; //store adapter in an array
    Find next innermost matched X/L patterns XL1 and XL2;
  }
  return createSPAdapter(P1, P2, blockAdapter);
}

```

**Figure 7:** Adapter creation Algorithm (PBA)

We first discuss the treatments of the *choice* and *loop* patterns. We start with the innermost *loop* or *choice* patterns in each process and create an adapter for this structure. Then we find the next innermost loop or choice structure and find an adapter for it, and so on. While matching *choice* patterns, we create individual adapters for each unique pair of branches by invoking the `createSPAdapter` algorithm (see Section 4.4), and then combine these adapters in a *choice* pattern themselves (see Figure 8(a)). For *loop* patterns, forward branches and backward branches are respectively matched, and the created adapters are put inside a *loop* as shown in Figure 8(b). After an adapter is created for a *choice* or *loop* pattern, this subprocess is replaced by a single block with a new block id in the parent process. The matching block id's and their adapters are stored in an array for later reference. This procedure is repeated for all the *choice* or *loop* patterns, until an adapter is created for the outermost one. Thus, the remaining process will be left with only *sequence* and

*parallel* patterns (and blocks with ids) to which the SP adapter algorithm is applied again and a new adapter is returned. If the returned adapter is null, the two processes are compatible without the need of adapter. Note that function `findMatchedBranchPair` matches corresponding branches of the two (sub)processes represented by its arguments and returns one matching branch pair at a time. The match must ensure that each activity in one branch has a dual in the other branch.



**Figure 8:** Making adapters for choice and loop structures

#### 4.4 SP Adapter Creation Algorithm

For ease of presentation, we discuss the `createSPAdapter` algorithm separately. This algorithm is shown in Figure 9 and assumes that the two processes contain only S and P patterns. It works by simulating the execution of the two processes to create an adapter for S/P patterns. It starts with two lists of current activities (`currList1` and `currList2`) that are ready to run in the two processes (`p1` and `p2`), and a `pool` that is initially empty. It matches any send-receive pair or block pair in the current lists and removes them. If they are blocks, it adds their adapter which is already stored in an array to a set name `blockAdapterSet`. Then, if there is any entry in the `pool` that matches one in a current list, the pair is removed. If the entries are atomic then they are added to the `adaptActivitySet` set. If they are blocks, their adapter will be put in `blockAdapterSet`. Similarly, if there are any unmatched send activities or blocks in a current list, they are added to the `pool` when no activity is matched in a given round. A send entry is also added to the `adaptActivitySet`. Then the entries in `adaptActivitySet` are combined in parallel and added in sequence to the adapter being created. The algorithm runs until the current lists are empty. Last, the created adapter and all the adapters in `blockAdapterSet` are put inside a parallel pattern to make the final adapter, which is also a process itself. For instance, Adapter 1 in Figure 1 is described as:

**Par**(Seq(*receive ship info*, *send ship info*), Seq(*receive invoice*, *send invoice*))

#### 4.5 A Heuristic Algorithm for a Minimal Adapter (PBA-MIN)

Above we described algorithm PBA for creating an adapter. Next, we modify this algorithm and improve it further by raising the question, what is the smallest adapter we can create? Let us reconsider the example of Figure 1. As noted earlier, Adapter 2

is an improvement upon Adapter 1 because it only buffers *ship info*. After *ship info* is received by Adapter 2, the retailer process can receive the *invoice* from the manufacturer process. Then, the two processes transact *payment*. In the last step, the Adapter2 sends *ship info* to the manufacturer process. Thus, Adapter2 has only two activities in a *sequence* pattern, and is simpler than Adapter1.

```

Adapter createSPAdapter(Proc P1, Proc P2, array blockAdapter[] [])
{
  //Initialize structures
  pool={}; adapter={}; doneList1={}; doneList2={};
  currList1={child activities that are ready to execute in P1};
  currList2={child activities that are ready to execute in P2};
  adaptActivitySet={}; //set for activities to be added in this round
  blockAdapterSet = {}; // set will contain required block adapters

  repeat{
    foreach (a1 in currList1, a2 in currList2) // match current lists
    {match activities and blocks in currentlist1 with currentlist2:
      if (match found){remove from current list; add to done list;}
      if two blocks match
      {add adapter for matching blocks to blockAdapterSet;}
    }

    foreach (a1 in {currList1, currList2}, a2 in pool) //Match w/pool
    {match activities and blocks in currentlists with pool s.t.:
      if (match found)
      {remove them from currentlist and pool;
        add to doneList1, doneList2;
        add dual(a2) to adaptActivitySet;
      }
      if (two blocks match)
      {add adapter for matching blocks to blockAdapterSet;}
    }

    foreach (a in {currList1, currList2}) //Add unmatched to pool
    {if ((a.mode == 'send') || (a.type != null))
      {add a to pool; remove a from currList; add a to doneList;
        add dual(a2) to adaptActivitySet;
      }
    }

    adapter = Seq(adapter, Par(adaptActivitySet));
    //put activities in adaptActivitySet in Par pattern and append
    adaptActivitySet = {}; // reinitialize adaptActivitySet now

    update currList1 with successors of doneList1 in P1;
    update currList2 with successors of doneList2 in P2;
    doneList1={}; doneList2={}; //reinitialize donelist1, doneList2
  } while (currList1 != null || currList2 != null); // end outer loop

  adapter = Par(adapter U blockAdapterSet);
  //put the adapters for Blocks in Parallel with the current adapter
  return adapter;
}

```

**Figure 9:** SP Adapter Creation Algorithm

In order to get a minimal adapter, intuitively we should buffer as few messages as possible. Consequently, in each iteration of the SP adapter algorithm (Figure 9), if both processes are stuck, we can buffer only one send activity. After this send activity is buffered, if both processes are still stuck, we would buffer another send activity in the next iteration, and so on, until the processes are free.

To implement the above idea, we need to choose the "best" send activity to buffer, i.e. one which has the highest probability to resolve the block. First, we define the *distance* between two activities simply as the number of edges or steps in a shortest path connecting them (it is infinity if there is no path). In Figure 1 for example, we observe that when both processes are stuck, i.e. the retailer process stops at *send ship info* and the manufacture process stops at *send invoice*, the distance between *send ship info* and *receive invoice* (the dual of *send invoice*) is 1, while the distance between *send invoice* and *receive ship info* (the dual of *send ship info*) is 2. Intuitively, if we buffer *ship info*, it only takes one step to free the stuck status of the manufacturer process, while it takes two steps to free the stuck status of the retailer process if we buffer the *invoice*. Therefore, we buffer the *ship info* first. To formalize this idea, we define the *minimum receive distance* next.

**Definition 1. (Minimum Receive Distance):** The *minimum receive distance* (MRD) of a send activity in a current list of a process is the minimum of the distances between this activity and each of the receive activities, which are the duals of the send activity/activities in the current list of the other process, i.e.,

$$\begin{aligned} \text{MRD}(a_1) &= \text{Min}(\text{Dist}(a_1, \text{Dual}(a_2))), \\ &\text{where } (a_1 \in \text{currList1} \ \&\& \ a_1.\text{type} == \text{'send'} \ \&\& \\ &\quad a_2 \in \text{currList2} \ \&\& \ a_2.\text{type} == \text{'send'}) \\ &|| (a_1 \in \text{currList2} \ \&\& \ a_1.\text{type} == \text{'send'} \ \&\& \\ &\quad a_2 \in \text{currList1} \ \&\& \ a_2.\text{type} == \text{'send'}) \end{aligned}$$

The smaller the MRD of a send activity, the larger is the probability that the buffering of this activity can free the processes. Therefore, a reasonable heuristic (**Heuristic 1**) is to buffer the send activity ( $a_1$ ) with the smallest MRD when both processes are stuck and both current lists contain send activities, i.e.

$$\text{selectedSend} = a_1, \text{ s.t. } \text{MRD}(a_1) = \text{Min}\{\text{MRD}(a)\}$$

In the example of Figure 1, both current lists contain at least one send activity. However, in some other cases, when two processes are stuck, it could also be that only one current list (say, *currList1*) contains all send activities, while the other current list (say, *currList2*) may contain only receive activities. In such situations, to buffer the send activity closer to the send activities which are the duals of the receive activities in *currList2* is more likely to free the stuck processes. To formalize this idea, we define the *minimum send distance* next.

**Definition 2. (Minimum Send Distance):** The *minimum send distance* (MSD) of a send activity in the current list of a process is the minimum of the distances between this activity and each of the activities which are the duals of the receive activities in the current list of the other process, i.e.,

$$\begin{aligned} \text{MSD}(a_1) &= \text{Min}(\text{Dist}(a_1, \text{Dual}(a_2))) \\ &\text{where } (a_1 \in \text{currList1} \ \&\& \ a_1.\text{type} == \text{'send'} \ \&\& \\ &\quad a_2 \in \text{currList2} \ \&\& \ a_2.\text{type} == \text{'Recv'}) \\ &|| (a_1 \in \text{currList2} \ \&\& \ a_1.\text{type} == \text{'send'} \ \&\& \\ &\quad a_2 \in \text{currList1} \ \&\& \ a_2.\text{type} == \text{'Recv'}) \end{aligned}$$

The smaller the MSD of a send activity is the greater is the probability that buffering this activity can free the stuck processes. Therefore, a second heuristic (**Heuristic 2**) is to buffer the send activity ( $a_1$ ) with the smallest MSD when both processes are stuck and only one current list contains send activities, i.e.

$$\text{selectedSend} = a_1, \text{ s.t. } \text{MSD}(a_1) = \text{Min}\{\text{MSD}(a)\}$$

Our algorithm is shown in Figure 10. For brevity, we only list the treatment for the send activities in both current lists, where the major difference from the SP adapter algorithm in Figure 9 lies. The algorithm maintains a variable *isStuck* to indicate the status of the two processes. Any time both processes are stuck, the SP algorithm sets *isStuck* to True. Then the above two heuristics are applied depending upon whether both current lists contain send activities (Heuristic 1), or only one contains send and the other only receive activities (Heuristic 2). In the two cases the send activity with the minimum MRD or the minimum MSD, respectively, is selected for adding to the adapter, and the variable *isStuck* is set to False.

```

if (isStuck)
{
  sendList1 = getSendActivity(currList1);
  sendList2 = getSendActivity(currList2);
  if ((sendList1 is empty) || (sendList2 is empty))
  //if only one currList contains send activity
  {Calculate MSD for each send activity in non-empty list;
  Select the send activity a with the smallest MSD;
  } else //if both currLists have send activities
  {Calculate MRD for each send activity in the sendLists;
  Select the send activity a with the smallest MRD;
  }
  add a to pool;
  remove a from currList1 or currList2;
  add a to doneList;
  add dual(a) to adaptActivitySet;
  isStuck = FALSE;
}

```

**Figure 10:** Algorithm for PBA-MIN adapter based on MRD and MSD calculations

There is a subtle semantic difference between the adapters created by PBA and PBA-MIN algorithms. The adapter created by PBA-MIN allows a send to get blocked if the corresponding receive of the other party is not ready, while ensuring there is no deadlock. For example, in Figure 1 while using Adapter 2 the manufacturer may get blocked while trying 'send Invoice' if the retailer has not performed the 'send Ship Info' activity first. However, this issue does not arise in Adapter 1 created by PBA because the 'send invoice' activity is also buffered in the adapter and so this send will not block. This is the cost of creating a smaller adapter.

## 5 Implementation, Validation and Evaluation

### 5.1 Testing and validation

An initial prototype system called BPCT (Business Process Composition Tool) was implemented in Java. To validate the correctness and evaluate the performance of our approach, we designed a Test Case Generator (TCG) to randomly create process pairs. For simplicity, we only considered the composition of two processes. *Process size* is the number of activities in a process. Given a specific process size *n*, TCG generates *n* messages, and for a message *i* a send activity ( $S_i$ ) and a receive activity ( $R_i$ ) are created. TCG randomly allocates  $S_i$  and  $R_i$  to two processes ( $P_a$  and  $P_b$ ). Therefore, Process  $P_a$  and  $P_b$  both have *n* activities each. For each process, all the activities are

put into a set. TCG randomly selects two activities from the set and assigns a composition pattern to them (e.g. sequence, parallel, choice, or loop). Then, these two activities are replaced by the composite activity in the set. TCG repeats these steps until there is only one activity left in the set, which is the root activity for the process. Thus, two random processes are constructed.

Considering the strict compatibility rules for *choice* and *loop* patterns, a totally random approach will generate a very large number of incompatible test cases. In order to improve the ratio of correct cases, we adjusted the ratio of sequence, parallel, choice and loop patterns to 4:4:1:1. Thus, the probability of choosing choice or loop pattern is reduced to 10%. Second, the iteration count of a loop pattern is set arbitrarily to either 5 or 10, with equal probability.

The Process Execution Simulator (PES) simulates the interaction between the processes. Given a set of processes ( $P_1, \dots, P_m$ ), PES puts the first activity (activities) of each process into an execution pool. Then, PES checks whether any two activities in this pool can interact with each other. If so, these two activities are deleted from the pool and their following activities are put into the execution pool. PES repeats these steps until no remaining pair of activities in the pool can be executed. If the execution pool is eventually empty, the composition succeeds. Otherwise, the composition fails. In PES, we only consider the synchronous communication model.

In PES, we have special treatments for the *choice* pattern. For the compatible cases, the branches of choice patterns have been matched with each other. We record this matching information and use it in the execution of choice patterns of PES. Suppose *choice* pattern  $Xor\_1$  in  $P_1$  and  $Xor\_2$  in  $P_2$  are duals of each other.  $Xor\_1$  has two branches  $B11$  and  $B12$ . And,  $Xor\_2$  has two branches  $B21$  and  $B22$ , such that  $B11$  is compatible with  $B21$  and  $B12$  is compatible with  $B22$ . If  $Xor\_1$  is executed before  $Xor\_2$ , PES will randomly choose  $B11$  or  $B12$  with equal probability. Assume  $B11$  is selected to execute. Then, when  $Xor\_2$  is ready to execute, PES will automatically select the appropriate branch in  $Xor\_2$  that is compatible with  $B11$ .

The validation test was conducted in three steps: 1) we used TCG to generate a test case; 2) without the compatibility checking, we ran this case in PES (with a result of success or failure); 3) we ran the compatibility verification and adapter creation algorithms on this case. The final result could be *incompatible*, *compatible-with-adapter* and *compatible-without-adapter*. The compatible-with-adapter cases were run with the generated adapter in PES. If the execution failed, the algorithm was invalidated. Besides, the algorithms were validated if the execution succeeded with the adapter that was produced by our algorithm. The validation test was repeated for, and passed, 40,000 test cases of process sizes with 5, 10, 20 and 50 activities.

## 5.2 Performance

One way to evaluate the performance of our adapter creation algorithms is to evaluate the complexity of the created adapters. Since the adapters themselves are business processes, we need evaluation metrics for a business process model. In this section, we used two major metrics of business process complexity from literature. Using these metrics, we ran the performance experiment on the Pattern-based Analysis (PBA), PBA-MIN, and the Service Execution Tree (SET) [5] methods. The results will be discussed shortly after introducing the metrics.

## NOAC Metric

The NOAC (Number of activities and control-flow elements in a process) metric gives the sum of the activities and the process control-flow elements in a process. Thus, we count the number of activities, and the control activity pattern pairs, a split-join pair being counted only once since we assume that the processes are well structured, and so are the adapters.

## CFC Metric

Following Cardoso [8], the CFC (Control-flow Complexity) metric is defined as follows (where  $P$  is a process and  $c$  is a control node in the process):

$$CFC(P) = \sum_{\substack{c \\ XOR-split}} CFC_{XOR-split}(c) + \sum_{\substack{c \\ AND-split}} CFC_{AND-split}(c) + 2 \sum_{\substack{c \\ LOOP-split}} c$$

where,  $CFC_{XOR-split}(c) = fan-out(c)$ , and  $CFC_{AND-split}(c) = 1$ .

In this metric, for an XOR (or OR-split) node with fan-out of  $n$ , exactly  $1$ -of- $n$  possible paths is taken in execution. Since this measure of complexity is based on the number of possible paths, every  $XOR-split$  with  $n$  outgoing transitions adds  $n$  to the CFC metric of this model. For a PAR (or  $AND-split$ ) node, with  $n$  outgoing activities, all the outgoing activities must be done. Therefore, every  $AND-split$  in a model adds  $1$  to the CFC metric. The LOOP pattern is treated similarly to an XOR-split since it can be implemented using a 2-way XOR-split and join; therefore every  $LOOP-split$  adds 2 to the metric.

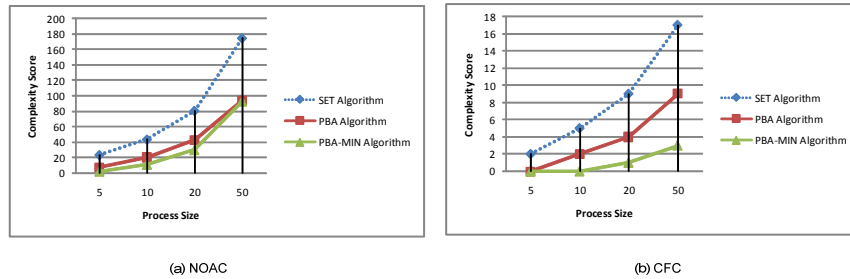


Figure 11: Performance comparison in terms of Process Complexity Metrics

To comprehensively evaluate the performance of the algorithms, we used TCG to create 500 non-deadlocked process cases of size 5, 10, 20 and 50. For each case, we generated the PBA adapter, PBA-MIN adapter and the SET adapter. Then, we calculated the NOAC and CFC complexity metrics on each of them. Figure 11 plots the average complexity metrics for each process size.

Figure 11 shows that for each process size, the average complexity of the PBA adapter is only half (or less than half) of the SET adapter complexity. Moreover, the results also show that for the NOAC metric, PBA-MIN has a slightly better performance than PBA. Furthermore, with the CFC metric, the PBA-MIN approach produces remarkable improvement. The CFC score of PBA-MIN is less than half of

PBA's CFC score, which means that although the PBA adapter and PBA-MIN adapter are relatively equivalent in terms of the numbers of activity and control elements, the complexity of the control elements in PBA-MIN is much less.

## 6 Discussion and Related Work

We showed how to check whether two or more interacting processes designed with sequence, parallel, choice and loop patterns (say, in WSBPEL [24]) can be combined to create composite processes. One limitation of our work is that we do not allow unstructured workflows, which is possible in WSBPEL with the use of links. Instead, we have focused our efforts on the four basic structures because they are the most common ones, and some, while not all, unstructured workflows can also be rewritten in terms of these basic structures. Our main idea is that formal pattern matching rules can be applied to reduce the search space of both verification and adapter creation algorithms. Using these rules, we develop a verification algorithm, and two adapter creation algorithms PBA and PBA-MIN. The PBA-min algorithm is based on two heuristics and we argue (without formal proof) that it creates a minimal adapter. The adapters are created, tested and validated, and evaluated for their complexity using process complexity metrics.

In the literature, many approaches [1,2,6,11,26] tackle the composition problem of Web services by identifying mismatches between service interfaces, syntactic differences among the exchanged messages, semantic mismatches among the exchanged messages, etc. The idea of matching structural patterns also appears in [19] but is not developed in any detail as done here. A closely related work to ours is that of Brogi and Popescu [5]. There the authors have given an approach to create adapters based on service execution trees (SET). However, they do not provide a full algorithm; their adapter is exhaustive (and much larger as we showed in Section 5.4) because it buffers every send activity; they do not consider all basic patterns; and, the approach leads to a combinatorial explosion with choice patterns because a service execution tree is generated for every path that can be taken. Thus, if there are  $n$  choice patterns in a process, there will be  $2^n$  unique service execution trees versus  $2n$  in our algorithm, assuming that no nested choice patterns are present. Another approach is given in [1,22]. It is quite comprehensive in that it focuses on both data and control flow mediation; however, it translates all process patterns into a finite state machine, which increases the computing complexity of the process composition. The approach in [30] is useful, but limited to finite state automaton and it does not allow parallelism.

Several other approaches for web services composition may be classified as: Logic based, Petri-net based, Pi-calculus based, State machine based, and Semantics based. In the logic-based method [20] process descriptions are written in a temporal logic language like Promela and then properties of the composite process are verified using a tool such as SPIN [14]. [16] gives an approach for building a controller for an open workflow net, as a graph with Boolean conditions for the nodes to determine feasible directions of progress. The Petri-nets approach [21] describes processes as Petri-nets and then applies Petri-net verification tools (such as Woflan [28]) to check properties like reachability, livelocks and deadlocks. Processes can also be described in Pi-calculus [3,18] and then their properties can be checked using a language like mu-logic. Some approaches [2,6,30] view a process as a state machine and combine these machines together to create a larger state machine for the composite process. Some of

these approaches lead to very large state spaces and can be computationally expensive. The pi-calculus approach can be very hard for end users to relate to. Finally, the semantics based methods such as OWL-S [25], WSMO [27] and Meteor-S [26] capture the semantics of web services and also use concept ontologies. A Conversational Support Model for business process integration is discussed in [13].

Interface adaptation is closely related to process adaptation. An interesting approach to interface adaptation is given in [9]. It is based on new operators like flow, gather, scatter, collapse, burst and hide. The notion of service views as means for adapting a web service is discussed in [11]. Service views are implemented using a scripting language. Semantic approaches for adaptation are discussed in [22]. A formal approach for reasoning about interaction patterns is given in [7,12]. Our work is complementary to these efforts since both kinds of adaptation are necessary.

## 7 Conclusions

We described algorithms that can test various processes (described, say in WSBPEL using four basic patterns) for compatibility and generate a minimal adapter. The algorithms are based on using a pattern compatibility matrix and matching rules to reduce the search space. We also did preliminary testing with a tool to validate the algorithm. Performance comparisons based on the complexity of the adapter produced by our two algorithms, PBA and PBA-MIN, and the SET algorithm [5] were reported and discussed.

Our focus here was on control flow mediation between processes. However, we can extend our matching rules to incorporate interface mismatches as well. Thus, when applying the matching rules one can also check whether any interface mismatch between 'send order' and 'receive order' can be resolved. For now we have assumed that there is no such mismatch. We also assumed that internal activities run in zero time. We would like to do experiments to see the effect of removing this assumption. Although we have argued informally that PBA-min gives a minimal adapter, yet a formal proof is required. Further, we plan to extend this approach to  $n$  interacting processes. Finally, we would like to develop our tool further and test it more rigorously, and also include other patterns (beyond sequence, choice, parallel and loop) into our algorithm for adapter creation.

## References

1. B. Benatallah, F. Casati, D. Grigori, H. R. M. Nezhad, and F. Toumani: Developing adapters for web services integration. Proceedings CAiSE 2005.
2. D. Berardi, et al.: Automatic composition of transition-based semantic web services with messaging. Proceedings of VLDB 2005: 613-624.
3. A. Bracciali, A. Brogi, C. Canal: A formal approach to component adaption. The Journal of Systems and Software, 74, 2005.
4. A. Brogi, C. Canal, E. Pimentel, A. Vallecillo: Formalizing web services choreographies. Electronic Notes in Theoretical Computer Science, 105, 2004.
5. A. Brogi, R. Popescu: Automated generation of BPEL adapters. Dan and Lamersdorf, editors, ICSOC'06, Vol. 4294 of LNCS, 27-39, Springer, 2006.
6. T. Bultan, X. Fu, R. Hull, J. Su: Conversation specification: a new approach to design and analysis of e-service composition. Proceedings International WWW 2003 Conference.

7. N. Busi, et al.: Choreography and orchestration conformance for system design. Proceedings of 8th Int'l Conference on Coordination Models and Languages, 2006.
8. J. Cardoso: Evaluating the process control-flow complexity measure. Proceedings IEEE International Conference on Web Services, 2005, 804-805.
9. M. Dumas, M. Spork, K. Wang: Adapt or perish: Algebra and visual notation for service interface adaptation. Proceedings BPM 2006, Vienna, Austria, September 2006.
10. X. Fu, T. Bultan, J. Su: Analysis of interacting BPEL web services. Proceedings of International WWW 2004 conference.
11. M. Fuchs: Adapting web services in a heterogeneous environment. Proc. IEEE International Conference on Web Services (ICWS'04), June 2004.
12. R. Gorrieri, C. Guidi, R. Lucchi: Reasoning About Interaction Patterns in Choreography. LCNS, No. 3670, 333-348, 2005.
13. J. Hanson, P. Nandi, S. Kumaran: Conversation support for business process integration. EDOC 2002: 65-74.
14. G. Holzmann: SPIN Model Checker. Addison-Wesley, 2004.
15. B. Kiepuszewski, A. H. M. Hofstede, C. Bussler: On structured workflow modeling. Proceedings CAiSE'2000, LNCS Vol. 1797, Springer Verlag.
16. N. Lohmann, et al. Analyzing interacting BPEL processes. Proceedings BPM 2006, Vienna, Austria, September 2006, 17-32.
17. J. Mendling, et al.: A Quantitative analysis of faulty EPCs in the SAP reference model. BPM Center Report, Eindhoven University of Technology, 2006.
18. R. Milner: Communication and concurrency. Prentice-Hall, 2004.
19. S. Moser, et al.: A hybrid approach for generating compatible WS-BPEL partner processes. BPM 2006, Vienna, Austria, September 2006, 458-464
20. S. Nakajima: Verification of web service flows with model-checking techniques. Proc. First international Symposium on Cyber Worlds (CW'02), Washington, DC, 0378.
21. S. Narayanan and S. McIlraith: Simulation, verification and automated composition of web services. Proceedings of the 11th Int'l WWW Conference (WWW '02).
22. M. Nezhad, et. al.: Semi-automated adaptation of service interactions. Proceedings of the 16th international WWW Conference (WWW '07), 993-1002.
23. E. Newcomer, G. Lomow: Understanding SOA with web services. Addison Wesley, 2005.
24. OASIS Web Services Business Process Execution Language (WSBPEL), Version 2.0, April 2007. Available at: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
25. OWL-S: Semantic Markup for Web Services, Available at: <http://www.daml.org/services/owl-s/1.2/overview/>.
26. A. Patil, S. Oundhakar, A. Sheth, K. Verma: Meteor-S web service annotation framework. Proceedings WWW 2004: 553-562.
27. M. Stollberg: Reasoning tasks and mediation on choreography and orchestration in WSMO. In Proceedings of the 2nd International WSMO Implementation Workshop (WIW 2005), Innsbruck, Austria, June 2005.
28. H.M.W. Verbeek, T. Basten, W.M.P. van der Aalst: Diagnosing workflow processes using Woflan. The Computer Journal, 44(4):246-279. British Computer Society, 2001.
29. W3C Candidate Recommendation: Web Services ChoreographyDescription Language Version 1.0, November 2005. Available at <http://www.w3.org/TR/ws-cdl-10/>.
30. D. M. Yellin, R. E. Strom: Protocol specifications and component adaptors. ACM Trans. on Programming Languages and Systems, Vol. 19, No. 2, March 1997, 292-333.