Minimal Protocol Adaptors for Interacting Services

R. Seguel, R. Eshuis, P. Grefen

Beta Working Paper series 318
Minimal Protocol Adaptors for Interacting Services

R. Seguel, R. Eshuis, and P. Grefen

Information Systems Group, School of Industrial Engineering, Eindhoven University of Technology, The Netherlands. {r.e.seguel,h.eshuis,p.w.p.j.grefen}@tue.nl

Abstract. In dynamic e-business, organizations collaborate in a just-in-time fashion using loosely coupled services. To ensure interoperability of the services, behavioral mismatches between their protocols need to be resolved in a fast and efficient way, which can be done with protocol adaptors. We present an efficient, automated method to construct (if possible) a minimal protocol adaptor with parallelism for two asynchronously communicating business protocols. A minimal adaptor only processes those messages that cause the mismatch, and has less message overhead at run-time than a non-minimal adaptor. Existing methods only build adaptors that are sequential, synchronous, or non-minimal. We show that the proposed method increases the efficiency of service adaption both at run-time and design-time.

Key words: Service Adaptation, Service Composition, Protocol adaptor, Minimal Adaptor, Behavioral mismatch

1 Introduction

The Service Oriented Architecture (SOA) enables interoperability between loosely coupled services offered by autonomous organizations that collaborate to reach a business goal by delivering a product or service. Since the complexity of delivered products/services increases while the life-cycles are getting shorter [1], the collaborations are not static but change continuously. For example, in dynamic service outsourcing [2] services collaborate in a just-in-time fashion: an organization outsources a non-core part of its business process, for instance logistics, to a partner that is selected at the last possible moment [3].

Each partner business process can be wrapped in a BPEL service [4] that specifies the interaction constraints of the process [2]. Since the interaction constraints of each collaborating partner are based on its own business process, the interaction constraints of the partners can mismatch easily. In that case, the partner services cannot interact properly. There are two kinds of mismatch: interfaces mismatch and behavioral mismatch. An interface mismatch is due to differences in the formats and specifications of messages exchanged. Such a mismatch can be resolved using schema mapping and transformation tools [5,6,7].
A behavioral mismatch occurs if two composed services reach a deadlock, since each service is waiting for a message from the other service.

In this paper, we focus on resolving behavioral mismatches for loosely coupled services, i.e. services that use asynchronous communication. The services communicate by exchanging WSDL [8] messages and each received message is stored in a FIFO queue, and consumed at a later point. The services reach a deadlock if the first message in each queue is unexpected according to the corresponding protocol [9]. Mismatches can be resolved by using adaptors. An adaptor ensures that two composed services terminate properly by receiving and reordering a set of messages from a service such that the messages are delivered in the order that the other service expects.

We present an efficient, automated method to build (if possible) a minimal adaptor for two asynchronous business protocols, expressed in BPEL, that have a behavioral mismatch. A minimal adaptor processes only those messages that cause the mismatch. Compared to an adaptor that processes all messages, a minimal adaptor has less message overhead and is less complex, as we show in Section 5. The minimal adaptor is compatible with the protocols using asynchronous communication, i.e. it ensures the messages are communicated in the right order, and thus the protocols can proceed exchanging messages and terminate properly without reaching deadlock. Both the input protocols and the constructed minimal adaptor can contain parallelism and loops.

The method consists of three steps. First, it finds the minimal set of messages needing adaptation. For that, it analyzes the protocol syntax to identify behavioral mismatches using heuristics, instead of calculating the combined states of the parallel protocols. Next, it derives the dependencies of the messages contained in the minimal set. Finally, it constructs the minimal adaptor that satisfies these dependencies to ensure that the protocols communicate the messages in the order that they expect. We show in Section 5 that the method runs in polynomial time.

Existing methods [6,9,10,11,12,13,14] build adaptors for sequential protocols. They suffer from the state explosion problem if the input protocols contain parallelism, i.e. the methods then take time exponential in the size of the protocols to generate an adaptor. Also, these existing methods build adaptors that process all messages exchanged by the protocols, even if only some messages cause a deadlock. This way, these adaptors have more process complexity, leading to more message overhead at run-time, as we show in Section 5. Moreover, most of the existing methods [6,9,11,12,13,14,15] only generate adaptors for protocols using synchronous communication, while we focus on asynchronous communication. Section 6 presents a detailed overview of related work.

The main contribution of this paper is an efficient, automated method to construct protocol adaptors for loosely coupled, asynchronously communicating services. Both the service protocols and the constructed adaptors can contain parallelism and loops. The method makes service adaptation more efficient at design-time and run-time, since the adaptor only processes those messages that
cause a deadlock, leading to less complex adaptors that have less message overhead compared to adaptors that process all messages.

The remainder of this paper is organized as follows. Section 2 presents a motivating example and Section 3 describes the behavioral mismatches of two protocols. Section 4 defines a method to construct a minimal adaptor in an efficient way. Section 5 describes the complexity of our method and presents some examples. Section 6 details related work and Section 7 presents the conclusions.

## 2 Motivating Example

We introduce our running example that we use to explain and illustrate the adaptation method. Figure 1 depicts two business protocols that interact using asynchronous communication: Protocol \( P \) represents a client service buying a flight ticket and protocol \( Q \) represents the travel agency service. In Figure 1, the message exchanges (interactions) between the protocols are represented by dotted arrows. Directions of dotted arrows indicate the send (source) and the receive (target) nodes. Note that message labels are not shown in the figure, they are implied by the names of send and receive actions.

We represent the business protocols of \( P \) and \( Q \) as BPEL [4] protocols. Figure 3 present a snippet of abstract BPEL code of the agency service \( Q \). To simplify the presentation we use service names instead of \texttt{partnerLinks} and \texttt{portTypes}. We use \texttt{invoke} for send nodes and \texttt{receive} for receive nodes.

Protocols \( P \) and \( Q \) reach a deadlock; for example, \( P \) sends message sequence \texttt{FlightSelected}, \texttt{PaymentType} and \texttt{ClientID}. \( Q \) first expects a message \texttt{ClientID}. If \( Q \) receives the messages in its FIFO queue in the order they were sent, then
...<sequence name="Sequence">
  <invoke name="SendFlightSelected"
    inputVariable="FlightSelected"
    outputVariable="FlightSelected"/>
  <invoke name="SendPaymentType"
    inputVariable="PaymentType"
    outputVariable="PaymentType"/>
  <flow name="Flow">
    <sequence name="Sequence">
      <invoke name="SendClientID"
        inputVariable="ClientID"
        outputVariable="ClientID"/>
      ...
      <invoke name="SendInvoice"
        inputVariable="Invoice"
        outputVariable="Invoice"/>
    </sequence>
  </flow>...

Fig. 2: BPEL code snippet of protocol P of Fig. 1

...<sequence name="Sequence">
  <receive name="RecClientID"
    variable="ClientID"/>
  <flow name="Flow">
    <receive name="RecFlightSelected"
      variable="FlightSelected"/>
    <receive name="RecPayment"
      variable="Payment"/>
    <invoke name="SendInvoice"
      inputVariable="Invoice"
      outputVariable="Invoice"/>
  </flow>...

Fig. 3: BPEL code snippet of protocol Q of Fig. 1

Q cannot consume the message ClientID since this is at the end of its queue. Therefore, Q reaches a deadlock, and as a result also P. A protocol adaptor can resolve this mismatch by receiving the messages FlightSelected, PaymentType and ClientID from P and delivering them in the order ClientID, FlightSelected and PaymentType to Q. Then, Q can consume each message from its queue without reaching a deadlock.

Existing methods [6,9,10,11,12,13,14] would adapt all (nine) interactions. Unlike existing methods, we only adapt those interactions that cause a deadlock. The minimal set of interactions needing adaptation are shown in bold in Figure 1: {FlightSelected, PaymentType, ClientID, Payment, TxOk, TicketInfo}. Compared to an adaptor that processes all nine interactions, the minimal adaptor reduces the message overhead by 33%, as we show in Section 5.

In the next section, we characterize behavioral mismatches. We use the characterization in Section 4 to construct a minimal adaptor.
3 Behavioral Mismatches

In this section, we show the formal description for the sequel of this paper; this description is partly based on concepts that we have developed in [15] for synchronously communicating services.

3.1 Protocol Trees

We specify business protocols in BPEL [4]. Each BPEL protocol (without links) specifies a tree. Leaves of the tree are the send and receive activities and internal nodes correspond to structured activities [16]. Formally, a protocol tree $P$ is a tuple $(N^+, N^-, C, ctype, child, rank, M, label, mult, r_0)$ where

- $N^+$ and $N^-$ are disjoint sets of receive nodes and send nodes, respectively. Let $N = N^+ \cup N^-$. 
- $C$ is a set of control nodes, used to specify ordering of messages.
- $ctype : C \rightarrow \{SEQ, AND, XOR\}$ is a function that assigns to each control node its type. A $SEQ$ node specifies sequential behavior, a $AND$ node parallel behavior, and a $XOR$ node exclusive (choice) behavior.
- $child \subseteq (N \cup C) \times C$ is a relation. If $x \in N \cup C$ and $y \in C$, then $(x, y) \in child$ if $x$ is a child of $y$. A send or receive node has no children.
- $rank : N \cup C \rightarrow \mathbb{N}$ indicates the ordering of children of nodes with type $SEQ$. The ranks of two nodes are only compared if the nodes share the same parent that has type $SEQ$. We require that two different nodes with the same parent have different ranks, and that for a node $n$ with $l$ children, for any child $c$ of $n$, $rank(c) \in \{0, \ldots, l-1\}$. Using an overloading of notation, we write $rank(n, i)$, where $0 \leq i \leq l-1$, to indicate the unique child $c$ of $n$ for which $rank(c) = i$.
- $M$ is a set of messages.
- $label : N \rightarrow M$ is a function labeling a send or receive node with a message. We require that there are not duplicate labels. Most BPEL protocols used in practice seem to satisfy this constraint. This constraint will be lifted in future work.
- $mult : N \cup C \rightarrow \{1, +\}$ specifies how many times each node and its subnodes are executed. Multiplicity + indicates a repeat loop from where we abstract the boolean exit condition. A repeat loop executes its subnodes at least once because its condition is checked after each execution ($XOR$-split and $XOR$-join). For technical reasons [16], we require that each loop node has either more than one child node or is a node. If a protocol contains a control loop node with one child, the loop can be “pushed down” to the child. By repeating this procedure, eventually either a node or a control node with more than one child is reached.
- $r_0 \in C$ is the node such that each node is the descendant of $r_0$. Node $r_0$ is the root of the tree.

Let $n$ be a node of protocol $P$. The function $children(n)$ is defined as \( \{ x \mid (x, n) \in child \} \). By $children^*$ we denote the reflexive-transitive closure
of children. If \( n \in \text{children}^*(n') \), we say that \( n \) is a descendant of \( n' \) and that \( n' \) is an ancestor of \( n \). In particular, each node is ancestor and descendant of itself.

To ensure that the \textit{children} function indeed arranges nodes in a tree, we require that each node has one parent, except node \( r_0 \), which has no parent. Next, we require that \( r_0 \) is ancestor of every node in \( N \cup C \). These constraints ensure that nodes are structured in a tree with root \( r_0 \). Leaves of the tree are the nodes. Internal nodes have type \textit{SEQ}, \textit{AND}, \textit{XOR}.

Let \( P, Q \) two protocol trees. We assume that \( P \) and \( Q \) have disjoint nodes, but shared messages, to model message exchanges between \( P \) and \( Q \). If \( n \in N_P \) and \( n' \in N_Q \) and \( \text{label}(n) = \text{label}(n') \), then we write \( n = n' \).

In asynchronous communication, protocol \( P \) sends a message \( m \) to \( Q \) and this stores \( m \) in a FIFO queue to consume it at a later point: a node \( s \in N_P \) sends \( m \) to \( r \in N_Q^+ \), such that \( s = r \). Therefore, protocols are compatible if they communicate messages, storing and consuming these messages from their FIFO queues without reaching a deadlock.

### 3.2 Behavioral Relations

We define relations between pairs of leaf nodes that capture behavioral aspects of the protocols.

For a set \( X \) of nodes, the least common ancestor of \( X \), denoted \( \lca(X) \), is the node \( x \) such that \( x \) is ancestor of each node in \( X \) and every other node \( y \) which is ancestor of each node in \( X \) is ancestor of \( x \): \( X \subseteq \text{children}^*(x) \) and for every \( y \in N \) such that \( X \subseteq \text{children}^*(y) \), we have that \( x \in \text{children}^*(y) \).

Since nodes are arranged in a tree, every set of node has a unique least common ancestor. Based on the notion of \( \lca \), we define some behavioral relations on nodes. If the least common ancestor \( \lca \) of nodes \( n, n' \) is \textit{SEQ}, then \( n \) is done before \( n' \), we have \( \text{rank}(c_n) < \text{rank}(c_{n'}) \), written \( n < n' \); if the \( \lca \) is \textit{AND}, then \( n \) and \( n' \) are done in parallel, written \( n \& n' \); if the \( \lca \) is \textit{XOR}, then either \( n \) or \( n' \) is done, written \( n \oplus n' \).

To model do-while loops, we abstract the boolean condition from the choice branch (\textit{XOR-split} and \textit{XOR-join}) assigning multiplicity (number of iterations) to the internal node that is entered and exited by the loop, and multiplicity 1 otherwise. This way, we have the following relations for two nodes in the same tree: \( n \otimes n' \) with \( \otimes \in \{<, \& \lor, <^+, \&^+, \lor^+ \} \) [15,16]. Relations without superscript mean they have multiplicity 1. Figure 4 shows a protocol \( T \) with a loop and its corresponding protocol tree with the behavioral relations on its nodes.

Using these definitions, we represent two protocols \( P \) and \( Q \) as protocols trees and the behavioral relations on their nodes. We define an interaction \( i = (s, m, r) \) as a pair of a send node \( s \) and a receive node \( r \); \( s \) sends a message \( m \) to \( r \), where \( s \) is a node in \( P \) and \( r \) is a node in \( Q \) or \( s \) is a node in \( Q \) and \( r \) is a node in \( P \).

We use the behavioral relations to identify possible behavioral mismatches. We analyze the relation between the send and receive nodes in each protocol to determine which causes a deadlock, comparing pairs of interactions. Since the relations are defined on the syntax, we avoid calculating the state space, which
makes our approach efficient for protocols containing parallelism. We evaluated the approach on the examples discussed in Section 5, and found that each actual deadlock corresponded to a potential behavioral mismatch and vice versa. In future work, we wish to analyze the formal correctness of the identified mismatches, using the FSM framework of Yellin and Strom [9] as basis.

Two protocols using asynchronous communication reach a deadlock if they cannot consume a message stored in the FIFO queue. For example, in Figure 5(a) $i_1 = (s_1, m_1, r_1)$, $i_2 = (s_2, m_2, r_2)$ and $s_1 < s_2$ in $P$ and $r_2 < r_1$ in $Q$. $P$ send $m_1$ followed by $m_2$. Next, $Q$ stores the messages in its queue in the order they are received. $Q$ reaches a deadlock since it cannot consume the message $m_2$ that is at the end of the queue. This mismatch is resolved if we ensure the messages are received by $Q$ in the right order. Then, $i_1$ and $i_2$ have to be adapted to avoid a deadlock. For instance, if only $i_1$ is adapted, then the adaptor sends the message $m_1$ to $Q$ while $P$ sends $m_2$, and thus $Q$ cannot consume $m_2$ if it first receives $m_1$.

Note that in Figure 5 the cases (b) and (c) do not need adaptation because $P$ and $Q$ send and receive a message from each other: there is no ordering mismatch. However, (d) is a deadlock that cannot be resolved because both protocols wait for a message that was not sent before from the other party.

In Figure 5(e) the protocols need adaptation if $Q$ first receives $m_2$ and then $m_1$. This way, $i_1$ and $i_2$ have to be adapted to avoid a deadlock. However, there is an exception to this mismatch; see Figure 5(f): if there is a receive node in $P$ in the same parallel block of $s_1$ and $s_2$, which is before $s_2$, and the corresponding send node in $Q$ is between $r_1$ and $r_2$: then $Q$ always receives the messages in order $m_1$, $m_2$, and thus the interactions $i_1$ and $i_2$ are compatible.

By analyzing choices, if $s_1$ $\vartriangleleft$ $s_2$ in $P$ and $r_1$ $\vartriangleleft$ $r_2$ in $Q$, then only one message is exchanged either $m_1$ or $m_2$, and thus the protocols are compatible; see Figure 6(a). However, if $s_1$ $\vartriangleleft$ $s_2$ in $P$ and $r_1$ $\odot$ $r_2$ in $Q$ with $\odot = \{<, \&\}$, then the protocols reach a deadlock because $P$ sends either a message $m_1$ or $m_2$ while $Q$ expects both messages in sequence or in parallel; see Figure 6(b) and (c). Next, interactions having nodes with type $\vartriangleright$ are incompatible with other interactions having nodes with type $<$ or $\&$. Thus, interactions containing nodes with type $\vartriangleright$...
are only compatible with interactions containing nodes with type $\triangledown$. Therefore, only protocol trees that have isomorphic choice branches can be adapted [15,17].

To summarize the analysis of behavioral mismatches, we build the Interaction Analysis Matrix (IAM) shown in Table 1. Note that the matrix analyzes the interactions of two protocols using asynchronous communication. A different matrix is needed for the synchronous case, which we have defined in [15].

The IAM matrix is spread over two tables. Table 1(a) describes the relations where send nodes $s_1, s_2$ are in $P$ and the receive nodes $r_1$ and $r_2$ are in $Q$. Table 1(b) describes the relations where a send node $s_1$ and a receive node $r_2$ are in $P$ and the nodes $r_1$ and $s_2$ are in $Q$. For instance, to analyze the protocols of Figure 5(a) we have that $i_1=(s_1, m_1, r_1)$ and $i_2=(s_2, m_2, r_2)$; the matrix entry
Fig. 7: Possible Mismatches for Relations $<^+$ and $\&^+$

Table 1: Interaction Analysis Matrix (IAM)

IAM[$s_1 < s_2$][$r_2 < r_1$] in Table 1(a) indicates that both $i_1$ and $i_2$ have to be adapted.

The complete IAM has in total 64 entries, but 32 symmetrical comparisons are omitted in Table 1. The matrix shows 15 compatible cases with “c”, 13 unresolvable cases with “d” and 4 adaptable cases with “$i_1, i_2$”. These adaptable cases identify the minimal set of interactions needing adaptation.

To analyze protocols containing loops, we restrict ourselves to loops with same multiplicity. This restriction will be lifted in a future work. The analysis
is similar to relations with multiplicity 1; see Figure 7. Next, for loop relations with multiplicity +, a similar matrix exists and it is shown in Table 2.

In the next section, we define a three step method to construct minimal adaptors for asynchronous communication.

4 A Method to Construct Minimal Adaptors

We define an efficient, automated method that consists of three steps. In the first step, it finds the minimal set of interactions using the IAM. In the second step, it derives a graph specifying the dependencies between messages nodes of the interactions needing adaptation. In the third step, it construct a minimal adaptor by generating a protocol tree that satisfies all the dependencies. The protocol tree is mapped to BPEL code. These three steps are described in the three subsections below.

4.1 Identifying the Minimal Set of Interactions Needing Adaptation

We explain the steps of the algorithm shown in Figure 8 to identify the minimal set of interactions needing adaptation. Although this algorithm is used in [15] for the synchronous case, here we use it with the IAM that we defined for asynchronous protocols.

First, we consider two protocol trees $P$ and $Q$ with the behavioral relations on their nodes. Next, we build the set $I$ of interactions between $P$ and $Q$; see line 2 in Figure 8. For example, in Figure 1 $P$ and $Q$ have nine interactions, indicated with dotted arrows; therefore, $I$ contains nine elements.

Next, we build the set $I_a$ that contains the interactions we identify that need adaptation. This way, we look up the action specified for every ordered pair $i_1=(s_1, m_1, r_1)$ and $i_2=(s_2, m_2, r_2) \in I$ in the IAM of Table 1; see lines 3-13 in Figure 8. For example, in Figure 1 if $i_1=(\text{SendFlightSelected}, \text{FlightSelected}, \text{RecFlightSelected})$ and $i_2=(\text{SendClientID}, \text{ClientID}, \text{RecClientID})$, then the interaction specified in IAM $[\text{SendFlightSelected} < \text{SendClientID}]$ $[\text{RecClientID} < \text{RecFlightSelected}]$ indicates that $i_1$ and $i_2$ have to be adapted, and therefore they are added to $I_a$.

This way, we compare all interactions of $I$ and the algorithm outputs the set $I_a$, which is the minimal set of interactions needing adaptation. If two interactions have incompatible control types or they wait for a message indefinitely,
1: procedure MatchingIAM\( (P, Q) \)
2: \[ I \leftarrow \{ (x, y) | (x \in N^P \land y \in N^Q) \lor (x \in N^Q \land y \in N^P) \} \land x = y \] 
3: for \((w, x) \in I\) do
4: for \((y, z) \in I\) do
5: if \(w \in N^P, y \in N^P\) or \(w \in N^Q, y \in N^Q\) then
6: \[ I_a \leftarrow I_a \cup \{ IAM[w \oplus w, y | x \oplus z, z] \} \]
7: else if \(w \in N^P, z \in N^P\) or \(w \in N^Q, z \in N^Q\) then
8: \[ I_a \leftarrow I_a \cup \{ IAM[w \oplus y, z | y \oplus y, x] \} \]
9: else
10: print "Deadlock"; exit 0
11: end if
12: end for
13: end for
14: return \(I_a\)
15: end procedure

Fig. 8: Algorithm to find the interactions needing to be adapted

Fig. 9: Protocols \(P\) and \(Q\)

then the algorithm determines that no adaptor can be generated. If \(I_a\) is empty, then \(P\) and \(Q\) are compatible and an adaptor is not needed.

By using this algorithm implemented in our tool [18] with the protocols of Figure 1 as input, we obtain six interactions needing adaptation; see Figure 9:
\[ I_a = \{ (\text{SendFlightSelected}, \text{RecFlightSelected}), (\text{SendPaymentType}, \text{RecPaymentType}), (\text{SendClientID}, \text{RecClientID}), (\text{SendPayment}, \text{RecPayment}), (\text{SendTxOk}, \text{RecTxOk}), (\text{SendTicketInfo}, \text{RecTicketInfo}) \} \]
4.2 Deriving Dependencies between Message Nodes of the Minimal Set

Once we identified the minimal set of interactions $I_a$, we construct a minimal adaptor that satisfies the dependencies between send and receive nodes and their dependencies inside the control flow of the original protocols.

The adaptor has to satisfy dependencies to ensure that $P$ and $Q$ communicate messages in the right order, i.e., the order they expect. These dependencies are identified to preserve the ordering of message nodes and ensure the right communication of messages. Thus, protocols consume the messages in the right order from the FIFO queues and the composition terminates properly.

To identify dependencies, we consider the control flow aspects of the original protocols ($P$ and $Q$) and message interactions between them, without explicit data flow, specifying a dependency graph. We use the definition of dependency graph given in [19] that can be seen as an extension of flow graphs [20, 21] with parallelism.

A dependency graph [19] is a directed graph in which the incoming and outgoing edges between the vertices indicate their dependencies. A vertex with more than one incoming/outgoing dependency is called resp. join/fork; a branching type either XOR or AND is assigned to the dependencies. We use dummy nodes labeled with the branching type to describe dependencies. For instance, in Figure 11 the nodes $P_{openXor1}$, $P_{closeXor1}$ are dummy nodes that describe branching type XOR. Note that $closeXor/openXor$ indicates resp. join/fork with type XOR; numbers in the labels only indicate an id.

To construct the minimal adaptor, we first construct a dependency graph containing all dependencies of all message nodes that belong to pairs in $I_a$, considering the control flow of the original protocols and interactions. Next, we use this dependency graph in Section 4.3 to generate the protocol of the minimal adaptor.

We construct the dependency graph as follows:
Dep. □ Graph □ of □ pruned □

Dep. □ Graph □ of □ pruned □

Fig. 11: Dependency Graphs of Pruned Protocols

(a) First, we prune the protocols trees $P$ and $Q$, removing those message nodes that do not belong to pairs in $I_\alpha$; see Figure 10.

(b) Next, we transform each pruned tree into a dependency graph by visiting every node of a pruned tree from its root:

- If a control node with type $XOR/AND$ is visited, then we add corresponding open and close dummy nodes with type $XOR/AND$ to the dependency graph. For example, protocol tree $P$ has one $XOR$, we add dummy nodes $P_{openXor1}$, $P_{closeXor1}$ with branching type $XOR$; see Figure 11.

- If a most nested message node is visited, then we add that node to the dependency graph. For instance, $invoke\_ClientID$, $receive\_TxOk$ are nested nodes of $P$ and added to its dependency graph; see Figure 11.

- If a control node $SEQ$ is visited, then we add edges between its children, adding an edge from the first child to the next in the sequence, and so on, until the last child in the sequence. Those children were already added to the dependency graph since they are nested. Next, if necessary, we add an edge from the open dummy $XOR/AND$ to the first child of a $SEQ$ node, and an edge from the last child of that $SEQ$ node to the corresponding close dummy $XOR/AND$. For example, we add an edge from $P_{invoke\_ClientID}$ to $P_{receive\_TxOk}$ in the dependency graph of $P$; see Figure 11. Next, we add the edge from $P_{openAnd5}$ to $P_{invoke\_clientID}$, and from $P_{receive\_TxOk}$ to $P_{closeAnd5}$.

- For a control node $XOR/AND$ that is the most nested, if there are not edges from the open dummy $XOR/AND$ to their children or from their children to the close dummy $XOR/AND$, then we add these edges. For instance, Figure 11 show the edges from $P_{openXor1}$ to $P_{openAnd13}$ and from $P_{closeAnd13}$ to $P_{closeXor1}$.
• Finally, we set the start node of the dependency graph as the first added vertex. For example, Figure 11 shows that \( P \_openXor1 \) is the start node of the dependency graph of \( P \), and \( Q \_openXor1 \) of \( Q \).

(c) Next, we construct an integrated dependency graph joining the dependency graphs of pruned protocols \( P \) and \( Q \) derived in step (b). For that, we add edges from a send node \( s \) to its corresponding receive node \( r \) for every pair \((s, r) \in I_a\). Figure 12 illustrates the integrated dependency graph by joining the graphs of Figure 11.

(d) Next, we identify the longest path for every pair of vertices in the integrated dependency graph, using the Floyd-Warshall algorithm [22]. The resulting matrix output by this algorithm allows us to identify redundant edges. An edge is redundant if its source and target vertices have branching type AND and there exists another path between its source and target vertices. We remove those redundant edges from the integrated dependency graph. The longest path always starts from one of the start nodes of the dependency graphs derived in (b), and thus we set the start node with the source of the longest path. We remove the start node that is not in the longest path. For example, in Figure 12 the longest path starts in \( P \_openXor1 \), and thus we remove \( Q \_openXor1 \), \( Q \_closeXor1 \) from the graph.

(e) Next, we obtain the minimal subgraph that contains all nodes that are in \( I_a \), reconstructing the longest paths, to ensure the adaptor satisfies all dependencies of nodes in the minimal set. This is a dependency graph because all nodes are connected. Finally, we change the operations name of every node in the dependency graph: every send operation is changed by receive, and vice versa. The resulting graph corresponds to the dependency graph of the adaptor. This dependency graph is the input for the next step in Section 4.3. Figure 13 shows the dependency graph of the minimal adaptor for the protocols of Figure 1.
4.3 Generating the Minimal Adaptor that Satisfies All the Dependencies

To generate the protocol of the minimal adaptor, we transform the dependency graph obtained in the previous step into BPEL code. For this, we use a previously published algorithm that maps a dependency graph into a BPEL tree [19]. This algorithm was implemented in the context of the CrossWork project [1] and it generalizes an existing algorithm [21] for structuring flow graphs, which correspond to sequential dependency graphs.

The BPEL tree that we obtain from the dependency graph of Figure 13 is shown in Figure 14. This protocol tree corresponds to the minimal adaptor of protocols of Figure 1. The BPEL code of the adaptor is automatically generated.
by our tool [18] and it is shown in Figure 15. The minimal adaptor is also depicted in a statechart in Figure 16. Note that the Figures 9–15 are part of the output of our tool.

In the next section, we show the complexity of our method and some examples, discussing the results.
5 Performance Analysis

We analyze the performance of the method in two different ways. First, we analyze the complexity of the algorithms included in the method. Second, we evaluate our method on examples from literature.
5.1 Complexity

Protocols $P$ and $Q$ have $|I| = n$ interactions between them and each protocol has $n$ message nodes. The minimal adaptor adapts $|I_a| = n_a$ interactions and $n_a \leq n$. Moreover, calculating the lca of every pair of nodes takes linear time [16].

A dependency graph is generated using a DFS approach [22] that runs in linear time $O(|V| + |E|)$ were $V = 2|I_a|$. The algorithm to compare every pair of interactions in the IAM runs in quadratic time [15]. The most expensive step is to remove redundant edges, which uses a Floyd-Warshall algorithm [22] that runs in $O(|V|^3)$ were $V = 2|I_a|$. Moreover, the algorithm to transform a dependency graph into a BPEL tree runs in almost linear time [19]. The worst case is $|I_a| = |I|$. Therefore, the complexity of the three step method to construct a minimal adaptor is $O(n^3)$.

5.2 Examples

We present a comparison of examples used in literature to show the minimal adaptors generated with our method. To the best of our knowledge, there are not examples in software/service adaptation literature that show protocols with more than four nodes, containing parallelism, and using asynchronous communication.

We use our running example and the healthcare example of [15]. Also, we use the examples of [11,12,13]; they present a example with three and four provider services and one client service. For each example, we compose the provider services in a flow in one service that interacts with the client service. Originally these examples are defined for synchronous communication and sequential protocols. Therefore, we use steps 2 and 3 of our own method to generate for the five examples non-minimal adaptors that use asynchronous communication. We compare those adaptors with the minimal adaptors, generated by using the complete method.

To compare the examples we consider the concurrency of the adaptors; we use metrics based on the control flow complexity (CFC) [23], the number of adapted messages of the protocols (NMN) with NMN = $|I_a|$, the total number of nodes (TNN) of an adaptor, the overhead reduction (OvR) of the minimal adaptor and the total overhead reduction (TOvR). The CFC, NMN and TNN metrics indicate design-time improvements and OvR and TOvR metrics measure run-time improvements.

We calculate OvR and TOvR as follows. An adaptor that contains all interactions has $2n$ messages nodes; the minimal adaptor has $2n_a$ message nodes. Therefore, the minimal adaptor reduces $\left(1 - \frac{2n_a}{2n}\right)\times100\%$ of the number of messages exchanged by the adaptor and the composed services (OvR).

Next, the original protocols and an adaptor containing all interactions exchange $2n$ messages in total; however, the protocols and a minimal adaptor exchange $2n_a$ plus $n - n_a$ (messages not needing adaptation) messages in total. Therefore, the total overhead reduction (TOvR) is $\left(1 - \frac{n + n_a}{2n}\right)\times100\%$ of messages exchanged by the composed services.
We have implemented our tool [18] in Java as proof-of-concept of our adaptation method. We have tested the examples in a PC with a Dual Core 1.60GHz CPU (Intel Atom) and 1GB RAM, getting each minimal adaptor in less than 1 second.

Example | Adaptor with all interactions | Minimal Adaptor | Improvement
--- | --- | --- | ---
NMN | TNN | CFC | NMN | TNN | CFC | OvR | TOvR
Figure 1 | 14 | 21 | 1 | 1 | 14 | 3 | 33% | 17%
Hospital [15] | 12 | 26 | 1 | 1 | 13 | 3 | 33% | 17%
eMuseum [11] | 26 | 54 | 4 | 3 | 9 | 1 | 85% | 42%
eRestaurant [12] | 17 | 1 | 1 | 1 | 75% | 38%
PDA [13] | 13 | 28 | 3 | 1 | 7 | 1 | 77% | 38%

Table 3: Comparison of example adaptors

Table 3 shows that our method has remarkable improvements at design-time and at run-time, making service adaptation more efficient: the minimal protocol adaptor has lower process complexity and reduces the overhead of messages exchanged by the composed services. The CFC, NMN and TNN scores of the minimal adaptor are half of the scores of the adaptor containing all interactions. This lower complexity of the minimal adaptor produces OvR scores of more than 33% and TOvR scores higher than 17% that are good improvements of the message overhead.

We assessed the utility of the method by checking the actual behavior of the example protocols, using an execution algorithm adapted from [17]; Figure 17 illustrates the execution of the adaptor and the protocols of our running example. For each example, each actual deadlock corresponded to a behavioral mismatch according to the matrix, and conversely, each behavioral mismatch according to the matrix corresponded to an actual deadlock. Next, for each minimal and non-minimal adaptor we used the execution algorithm to check that the adaptor resolves all deadlocks of the example protocols, using the execution algorithm. We found that all deadlocks were indeed resolved. We will formalize the correctness of the identified mismatches in future work, using the FSM framework of Yellin and Strom [9] as basis.

Finally, we apply our method to an example of [10], which is the only related work on protocol adaptors for services that also considers asynchronous communication. However, the adaptors constructed by [10] are not minimal. For the example used in [10], that method generates an adaptor that contains all the interactions: it has NMN of 6, TNN of 13 and CFC of 2. In contrast, the minimal adaptor generated with our approach for that example has NMN of 1, TNN of 2, CFC of 0 since it is sequential, OvR of 83% and TOvR of 42%.

In the next section, we describe the related work on generating protocol adaptors.
6 Related Work

There are several research efforts on constructing protocol adaptors for interacting services [6,10,11,12,13,14,15,17] based on the seminal work by Yellin and Strom [9].

We present in Table 4 a comparison of our work with related research, highlighting our contribution in the ‘IAM method’ column. A detailed overview can be found in [24].
Although [6,9,10,11,12,13,14] define approaches that are efficient for sequential protocols, these approaches are confronted with the state explosion problem for protocols containing parallelism, i.e. the methods then take time exponential in the size of the protocols to generate an adaptor. We analyze the protocol syntax to efficiently identify behavioral mismatches of protocols containing parallelism and loops with same multiplicity.

The methods of [6,9,10,11,12,13,14] build adaptors that are not minimal, and thus each message is processed by the adaptor. The idea of using minimal adaptors is due to Kumar and Shan [17]. However, they do not consider an adaptor for protocols using asynchronous communication.

The main contribution of our work is that we consider protocol adaptors for asynchronous communication that are minimal. In contrast, all methods defined in [6,9,11,12,13,14,15,17] construct adaptors for protocols using synchronous communication. Note that the adaptors generated by [10] are not minimal (cf. Section 5).

Like [15,17] we assume structured processes. This is not a limitation since standards like BPEL [4] and OWL-S [25] are mainly structured [19].

All adaptation methods consider only a pairwise analysis of protocols to construct an adaptor. We will extend our method in a future work to adapt more than two protocols.

In earlier work, we developed an adaptor generation approach for synchronous communication [15].

In the next section, we present the conclusions and further work.

### 7 Conclusions and Future Work

We have presented an efficient, automated method to construct a minimal adaptor for two business protocols containing parallelism and loops that use asynchronous communication. The method is defined for BPEL services, but is also applicable in the context of Service Component Architecture (SCA).

The minimal adaptor has lower process complexity and is more efficient than adaptors that are constructed by existing methods, since these adaptors process
all interactions, leading to higher message overhead. So, the minimal adaptor reduces the overhead of messages exchanged at run-time.

The presented method makes service adaptation more efficient at design-time (efficient construction) and at run-time (less message overhead). This characteristic is important for dynamic e-business with a just-in-time nature [3] where incompatible protocols have to be adapted fast to meet changing business goals.

There are several directions for future work. We are currently extending our method considering protocols with links and loops with different multiplicity. On the formal side, we will analyze the correctness of the method, using the FSM framework of Yellin and Strom [9] as basis.

Although in this paper we focused on behavioral mismatches, we can extend our method to deal with interface mismatching by using ontologies and tools to transform and map message schemas [5,6,7].

Also, we will identify the architecture requirements for dynamic service adaptation where services collaborate to meet just-in-time business goals. This way, we will extend our method to build an adaptor for more than two BPEL protocols in a Dynamic Business Network Process [26].

References

<table>
<thead>
<tr>
<th>nr.</th>
<th>Year</th>
<th>Title</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>318</td>
<td>2010</td>
<td>Minimal Protocol Adaptors for Interacting Services</td>
<td>R. Seguel, R. Eshuis, P. Grefen</td>
</tr>
<tr>
<td>317</td>
<td>2010</td>
<td>Teaching Retail Operations in Business and Engineering Schools</td>
<td>Tom Van Woensel, Marshall L. Fisher, Jan C. Fransoo</td>
</tr>
<tr>
<td>316</td>
<td>2010</td>
<td>Design for Availability: Creating Value for Manufacturers and Customers</td>
<td>Lydie P.M. Smets, Geert-Jan van Houtum, Fred Langerak</td>
</tr>
<tr>
<td>315</td>
<td>2010</td>
<td>Transforming Process Models: executable rewrite rules versus a formalized Java program</td>
<td>Pieter van Gorp, Rik Eshuis</td>
</tr>
<tr>
<td>314</td>
<td>2010</td>
<td>Ambidexterity and getting trapped in the suppression of exploration: a simulation model</td>
<td>Bob Walrave, Kim E. van Oorschot, A. Georges L. Romme</td>
</tr>
<tr>
<td>312</td>
<td>2010</td>
<td>Tales of a So(u)rcerer: Optimal Sourcing Decisions Under Alternative Capacitated Suppliers and General Cost Structures</td>
<td>Osman Alp, Tarkan Tan</td>
</tr>
<tr>
<td>311</td>
<td>2010</td>
<td>In-store replenishment procedures for perishable inventory in a retail environment with handling costs and storage constraints</td>
<td>R.A.C.M. Broekmeulen, C.H.M. Bakx</td>
</tr>
<tr>
<td>310</td>
<td>2010</td>
<td>The state of the art of innovation-driven business models in the financial services industry</td>
<td>E. Lüftenegger, S. Angelov, E. van der Linden, P. Grefen</td>
</tr>
<tr>
<td>308</td>
<td>2010</td>
<td>Effect of carbon emission regulations on transport mode selection in supply chains</td>
<td>K.M.R. Hoen, T. Tan, J.C. Fransoo, G.J. van Houtum</td>
</tr>
<tr>
<td>307</td>
<td>2010</td>
<td>Interaction between intelligent agent strategies for real-time transportation planning</td>
<td>Martijn Mes, Matthieu van der Heijden, Peter Schuur</td>
</tr>
<tr>
<td>306</td>
<td>2010</td>
<td>Internal Slackening Scoring Methods</td>
<td>Marco Slikker, Peter Borm, René van den Brink</td>
</tr>
<tr>
<td>304</td>
<td>2010</td>
<td>Practical extensions to the level of repair analysis</td>
<td>R.J.I. Basten, M.C. van der Heijden, J.M.J. Schutten</td>
</tr>
<tr>
<td>303</td>
<td>2010</td>
<td>Ocean Container Transport: An Underestimated and Critical Link in Global Supply Chain Performance</td>
<td>Jan C. Fransoo, Chung-Yee Lee</td>
</tr>
<tr>
<td>302</td>
<td>2010</td>
<td>Capacity reservation and utilization for a manufacturer with uncertain capacity and demand</td>
<td>Y. Boulaksil; J.C. Fransoo; T. Tan</td>
</tr>
<tr>
<td>300</td>
<td>2009</td>
<td>Spare parts inventory pooling games</td>
<td>F.J.P. Karsten; M. Slikker; G.J. van Houtum</td>
</tr>
</tbody>
</table>
Capacity flexibility allocation in an outsourced supply chain with reservation

An optimal approach for the joint problem of level of repair analysis and spare parts stocking

Responding to the Lehman Wave: Sales Forecasting and Supply Management during the Credit Crisis

An exact approach for relating recovering surgical patient workload to the master surgical schedule

An iterative method for the simultaneous optimization of repair decisions and spare parts stocks

Fujaba hits the Wall(-e)

Implementation of a Healthcare Process in Four Different Workflow Systems

Business Process Model Repositories - Framework and Survey

Efficient Optimization of the Dual-Index Policy Using Markov Chains

Hierarchical Knowledge-Gradient for Sequential Sampling

Analyzing combined vehicle routing and break scheduling from a distributed decision making perspective

Anticipation of lead time performance in Supply Chain Operations Planning

Inventory Models with Lateral Transshipments: A Review

Efficiency evaluation for pooling resources in health care

A Survey of Health Care Models that Encompass Multiple Departments

Supporting Process Control in Business Collaborations

Inventory Control with Partial Batch Ordering

Translating Safe Petri Nets to Statecharts in a Structure-Preserving Way

The link between product data model and process model

Inventory planning for spare parts networks with delivery time requirements

Co-Evolution of Demand and Supply under Competition
<table>
<thead>
<tr>
<th>Year</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>An Efficient Method to Construct Minimal Protocol Adaptors</td>
<td>R. Seguel, R. Eshuis, P. Grefen</td>
</tr>
<tr>
<td>2009</td>
<td>Coordinating Supply Chains: a Bilevel Programming Approach</td>
<td>Ton G. de Kok, Gabriella Muratore</td>
</tr>
<tr>
<td>2009</td>
<td>Inventory redistribution for fashion products under demand parameter update</td>
<td>G.P. Kiesmuller, S. Minner</td>
</tr>
<tr>
<td>2009</td>
<td>Comparing Markov chains: Combining aggregation and precedence relations applied to sets of states</td>
<td>A. Busic, I.M.H. Vliegen, A. Scheller-Wolf</td>
</tr>
<tr>
<td>2009</td>
<td>Separate tools or tool kits: an exploratory study of engineers’ preferences</td>
<td>I.M.H. Vliegen, P.A.M. Kleingeld, G.J. van Houtum</td>
</tr>
<tr>
<td>2009</td>
<td>An Exact Solution Procedure for Multi-Item Two-Echelon Spare Parts Inventory Control Problem with Batch Ordering</td>
<td>Engin Topan, Z. Pelin Bayindir, Tarkan Tan</td>
</tr>
<tr>
<td>2009</td>
<td>Similarity of Business Process Models: Metrics and Evaluation</td>
<td>Remco Dijkman, Marlon Dumas, Boudewijn van Dongen, Reina Kaarik, Jan Mendling</td>
</tr>
<tr>
<td>2009</td>
<td>Restricted dynamic programming: a flexible framework for solving realistic VRPs</td>
<td>J. Gromicho; J.J. van Hoorn; A.L. Kok; J.M.J. Schutten</td>
</tr>
</tbody>
</table>

Working Papers published before 2009 see: [http://beta.ieis.tue.nl](http://beta.ieis.tue.nl)