An efficient method to construct minimal protocol adaptors
Seguel Pérez, R.E.; Eshuis, H.; Grefen, P.W.P.J.

Published: 01/01/2009

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author’s version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
An Efficient Method to Construct Minimal Protocol Adaptors

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. Two composed interacting services reach a deadlock if their business protocols have behavioral mismatches. A protocol adaptor can resolve deadlocks. However, existing methods build adaptors that process all messages exchanged by the protocols, even if only some messages cause a deadlock.

We present an efficient, automated method to construct (if possible) a minimal adaptor for two business protocols containing parallelism and loops. First, the method finds the minimal set of messages exchanged needing adaptation, using behavioral relations on the protocol syntax to identify mismatches. Next, it generates in an efficient way an adaptor from the minimal set of messages. This minimal adaptor is compatible with the protocols, it reduces process complexity and it improves runtime performance of the automated service composition. We have implemented the method in a tool for adapting two business protocols. We apply it to an example case study from the healthcare domain.

Key words: Service Adaptation, Service Composition, Protocol adaptor, Process Integration, Cross-organizational Processes

1 Introduction

Service Oriented Architecture (SOA) uses automated service composition and service coordination to integrate cross-organizational information systems between companies. A service can be very simple like one which converts an amount of money to another, or very complex like one that invokes complex business applications [5].

Composition languages like BPEL [2] enable the execution of cross-organizational processes, invoking web service operations rather than conventional applications [1]. BPEL is the standard language for business protocols to specify the order in which the composed services send and receive messages. In the service execution setting, two composed services reach a deadlock if their business protocols have behavioral mismatches. In that case, a protocol adaptor can be used to resolve deadlocks to ensure the composed services terminate properly.

To illustrate this, Figure 1 depicts two business protocols that interact using synchronous communication; Protocol $P$ represents a client service buying...
a flight ticket and protocol $Q$ represents the travel agency service. Message exchanges (interactions) between the protocols are represented by dotted arrows. Directions of dotted arrows indicate the sending (source) and the receiving (target) nodes. Protocols $P$ and $Q$ reach a deadlock; for example, $P$ sends a message FlightSelected that $Q$ expects later since it first expects a message ClientID. A protocol adaptor can resolve this mismatch by receiving the message FlightSelected from $P$ and delivering it only when $Q$ accepts that message. Note that message labels are not shown in the figure, they are implied by the names of sending and receiving actions.

Existing methods [3,7,11,12] build adaptors that process all messages exchanged by the protocols, even if only some messages cause a deadlock. For example, for protocols in Figure 1 these methods generate an adaptor containing all messages. However, we could construct an adaptor with only the messages needing adaptation to reduce the process complexity and improve run-time performance. In this way, the other interactions are not processed by the adaptor, and thus, the overhead of adaptation at run-time is reduced. For example, the minimal set of interactions needing adaptation are highlighted in Figure 1: FlightSelected, PaymentType, Invoice, and TicketInfo.

In this paper, we present an efficient, automated method to construct a minimal adaptor for two business protocols containing parallelism and loops. If the
business protocols are compatible, the method does not generate an adaptor. The minimal adaptor is compatible with the protocols using synchronous communication. This means that a party making a synchronous call cannot proceed until it receives a reply. Because the scope of this paper, we leave asynchronous communication and interface adaptation [8] as future work.

The method consists of two steps. In the first step, it finds the minimal set of interactions needing adaptation, using behavioral relations on the protocol syntax to identify mismatches. We compare pairs of message exchanges (interactions) to find exactly those needing adaptation instead of calculating the combined states of the parallel protocols. By comparing two interactions, we analyze the relation between the sending and receiving nodes of each protocol to determine which causes a deadlock. This step has quadratic performance.

In the second step, we build in an efficient way a generic parallel adaptor from the minimal set of interactions needing adaptation. For protocols of Figure 1, the minimal adaptor is shown in Figure 2. This minimal adaptor has better performance at run-time in terms of process complexity than an adaptor containing all messages.

The contribution of this paper is an efficient, automated method to construct minimal adaptors for service composition. This method resolves deadlocks to ensure the system executes properly. Using a minimal adaptor, it reduces the number of messages exchanged by the composed services at run-time. Next, this method makes service adaptation more efficient, which is a key capability to enable the Service Oriented Architecture (SOA) paradigm.

The remainder of this paper is organized as follows. Section 2 presents the behavioral relations for analyzing the interactions of two protocols. Section 3 shows a method to identify behavioral mismatches. Section 4 describes a method to construct a minimal adaptor in an efficient way. Section 5 shows an example case.
and Section 6 details related work. Finally, Section 7 presents the conclusions and further work.

2 Preliminaries

In this section, we use the definitions described in [4]. We represent a protocol as a tree and define behavioral relations and operations on protocol trees. The method in Section 3 uses these relations to analyze protocol interactions in an efficient way.

2.1 Protocol Tree Definitions

Cross-organizational processes can be executed using composition languages like BPEL[2]. Business protocols are represented by BPEL protocols. Each BPEL protocol (without links) specifies a tree and leaves of the tree are the basic activities while internal nodes correspond to structured activities.

Formally, a protocol tree $P$ is a tuple $(M^+, M^-, C, children, ctype, mult, root)$ where

- $M^+$ and $M^-$ are the receive messages and send messages, respectively. Let $M = M^+ \cup M^-$. 
- $C$ is a set of control nodes, used to specify ordering of messages. 
- $children : C \rightarrow P(C \cup M)$ is a function that defines for each control node its set of child nodes. A message node has no children. 
- $ctype : C \rightarrow \{SEQ, AND, XOR\}$ 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. 
- $mult : M \rightarrow \{1,^\ast\}$ specifies how many times each node and its subnodes is executed. Multiplicity $^\ast$ indicates a while-do loop. For technical reasons [4], we require that each loop node has either more than one child node or is a message 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 message node or a control node with more than one child is reached. 
- $root$ is the node such that each node is the descendant of $root$. 

For message nodes, we introduce some special notation. If $n, n'$ are two message nodes with $n = n'$ and $n \in M^+$ and $n' \in M^-$, then $n = \overline{n'}$.

Let $n \in M$ be a node of protocol $P$. By $children^*$ we denote the reflexive-transitive closure of $children$. If $n \in children^*(n')$, 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 $children$ function indeed arranges nodes in a tree, we require that each node has one parent, except one node $r$, which has no parent. Next, we require that $r$ is ancestor of every node in $M$. These constraints ensure
that nodes are structured in a tree with root \( r \). Leaves of the tree are the message nodes. Internal nodes have type \( SEQ, AND, XOR \).

To indicate the ordering of children of nodes with type \( SEQ \), we use a partial function \( \text{rank} : M \rightarrow \mathbb{N} \). 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 \( n \), \( \text{rank}(c) \in \{0, \ldots, l - 1\} \). Using an overloading of notation, we write \( \text{rank}(n, i) \), where \( 0 \leq i \leq l - 1 \), to indicate the unique child \( c \) of \( n \) for which \( \text{rank}(c) = i \).

To illustrate this definition, Figure 3 depicts protocol \( P \) of Figure 1 represented as a protocol tree. The statechart on the left is equivalent to the tree with structured (\( SEQ, AND, XOR \)) and message (leaves) nodes.

### 2.2 Behavioral Relations on Protocols

To efficiently analyze protocol interactions, we define functions on the syntax of protocol models considering the structure of protocol trees.

For a set \( X \) of nodes, the least common ancestor of \( X \), denoted \( \text{lca}(X) \), 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 \( y \in M \) 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. Sequential behavior is specified using SEQ nodes. SEQ nodes induce the < relation. Given two nodes \( n, n' \in M \), we have \( n < n' \) if and only if node \( l = \text{lca}\{n, n'\} \) has type SEQ, and for the children \( c_n, c_{n'} \) of \( l \) such that \( n \) is descendant of \( c_n \) and \( n' \) is descendant of \( c_{n'} \), we have \( \text{rank}(c_n) < \text{rank}(c_{n'}) \).

Next, we define relations for choice and parallelism. Given two nodes \( n, n' \in M \), we have

- \( n \lor n' \) if and only if node \( l = \text{lca}\{n, n'\} \) has type XOR.
- \( n \land n' \) if and only if node \( l = \text{lca}\{n, n'\} \) has type AND and there is no (in)direct control flow link connecting \( n \) and \( n' \), so not \( n < n' \).

The three defined relations are independent of loops, since they do not take multiplicity information into account. To model loops, we superscribe the three relations with multiplicity constraints of the involved least common ancestors. For example, \( a <^1 b \) indicates that \( \text{mult}(\text{lca}(a, b)) = 1 \) while \( a <^\ast b \) indicates \( \text{mult}(\text{lca}(a, b)) = \ast \). This way, we can distinguish a protocol with a loop from the same protocol with the loop removed. Unsuperscribed relations can have any multiplicity, so for example \( a < b \) means either \( a <^1 b \) or \( a <^\ast b \).

We define an interaction to analyze two protocol trees as follows. Let \( P \) and \( Q \) be two protocols. An interaction \( i = (s, r) \) is a pair of a sending message node \( s \) and a receiving message node \( r \); \( s \) sends a message to \( r \), \( s \in M^− \) and \( r \in M^+ \). For example, Figure 1 shows nine interactions with dotted arrows.

In the following section, we analyze protocol interactions to identify mismatches in a standard way. Next, we use this analysis in Section 4 to efficiently construct a minimal protocol adaptor.

### 3 Identifying Interaction Mismatches

We compare pairs of interactions to identify behavioral mismatches instead of calculating the combined states of the parallel protocols. We compare pairs because we can analyze the relation between the sending and receiving nodes in each protocol to determine which causes a deadlock, using behavioral relations.

To standardize this method, we build the Interaction Analysis Matrix (IAM) shown in Table 1. We use this matrix to compare every pair of interactions \( i_1 \) and \( i_2 \), analyzing the behavioral relations between their message nodes. With this matrix, we classify the message nodes according to the types and match their relations. Next, we determine which interactions need to be adapted and which cannot be resolved.

The matrix is constructed in two tables. Table 1(a) describes the relations where sending nodes \( s_1, s_2 \) are in \( P \) and the corresponding receiving nodes \( r_1 \) and \( r_2 \) are in \( Q \). The symmetrical case in which nodes \( r_1, r_2 \) are in \( P \) and \( s_1, s_2 \) are in \( Q \) is similar to Table 1(a) and therefore, omitted. Table 1(b) describes the relations where a sending node \( s_1 \) and a receiving node \( r_2 \) are in \( P \) and the corresponding nodes \( r_1 \) and \( s_2 \) are in \( Q \). This table describes 16 relations that
are similar to the symmetrical case in which nodes $s_2, r_1$ are in $P$ and $s_1, r_2$ are in $Q$, and thus not shown. Therefore, the IAM represented in Table 1 has in total 64 entries, but 32 symmetrical comparisons are omitted.

Entries with a “c” character in the IAM mean that $i_1$ and $i_2$ do not need to be adapted because the interactions are compatible. Entries with a “d” character represent that interactions have incompatible control types or both protocols wait for a message indefinitely. Entries with “i”, “i2”, or “i1,i2” determine which interactions need to be adapted because of a deadlock.

The matrix shows 16 compatible cases, 13 unresolvable cases and 3 adaptable cases. The adaptable cases are very relevant because we can easily classify, match and determine the mismatches between two interacting services using synchronous communication. We leave the extension of this matrix for the asynchronous semantics as further work because of the scope of this paper.

We first explain the entries in IAM comparing nodes with types $<$ and $&$. Next, we describe entries by comparing nodes with types $\triangledown$ and types $<$ and $&$. After that, we describe comparisons between nodes with types $\triangledown$. Finally, we describe loops.

**Sequence and Parallelism.** The analysis of two interactions $i_1$, $i_2$ with relations $<$ and $&$ in Table 1 is illustrated in Figure 4 and described as follows:

(a) If $s_1 < s_2$ in $P$ and $r_2 < r_1$ in $Q$, then $i_1$ has to be adapted because $P$ sends a message $m_1$ that $Q$ cannot receive since it first expects a message $m_2$; see Figure 4(a). The other case $s_2 < s_1$ in $P$ and $r_1 < r_2$ in $Q$ is symmetrical such that $i_2$ must be adapted; see Table 1(a). In contrast, if $s_1 < s_2$ in $P$ and $r_1 < r_2$ in $Q$ or $s_2 < s_1$ in $P$ and $r_2 < r_1$ in $Q$, then these interactions are compatible and illustrated with “c” in Table 1(a). If $s_1 < s_2$ or $s_2 < s_1$ in $P$ and $r_1,r_2$ in $Q$, then these interactions are compatible because protocol $Q$ expects the messages $m_1$ and $m_2$ in parallel. Also, these matchings are illustrated with “c” in Table 1(a).

(b) If $s_1 < r_2$ in $P$ and $s_2 < r_1$ in $Q$, both $P$ and $Q$ send a message before receive from each other (see Figure 4(b)), then both $i_1, i_2$ have to be adapted as it is shown in Table 1(b). However, there are two other alternatives: adapting $i_1$ or $i_2$. If the synchronous semantics implements an arbitrator like [?] that gives to $P$ precedence to send $m_1$, then $i_1$ should be adapted; otherwise, $i_2$ has to be adapted. We only consider $i_1, i_2$ to be adapted; the other two...
alternatives are out of scope of this paper. In the symmetrical case \( s_1 < r_2 \) in \( P \) and \( r_1 < s_2 \) in \( Q \) the sequences are compatible and the entry with “c” is shown in Table 1(b).

(c) If \( r_2 < s_1 \) in \( P \) and \( r_1 < s_2 \) in \( Q \), both \( P \) and \( Q \) expect a message that was not sent before and both protocols wait for a message indefinitely; see Figure 4(c). This is denoted with “d” in Table 1(b). In the symmetrical case in which \( r_2 < s_1 \) in \( P \) and \( s_2 < r_1 \) in \( Q \), interactions are compatible and the entry is presented with “c” in Table 1(b).

(d) If \( s_1 \& s_2 \) in \( P \) and \( r_1 < r_2 \) in \( Q \), then the interactions are compatible because \( Q \) first waits for a message \( m_1 \) and next it accepts a message \( m_2 \), even though \( P \) sends both messages in parallel because of the synchronization messages; see Figure 4(d). The symmetrical cases where \( s_1 \& s_2 \) in \( P \) and \( r_2 < r_1 \) or \( r_1 \& r_2 \) in \( Q \) are also compatible. The corresponding entries are illustrated with “c” in Table 1(a).

(e) If \( s_1 \& r_2 \) in \( P \) and \( s_2 < r_1 \) in \( Q \) (see Figure 4(e)), it is similar to the case explained in (d). The other combinations of this case: \( s_1 \& r_2 \) in \( P \) and \( r_1 < s_2 \) in \( Q \), and \( r_2 < s_1 \) in \( P \) and \( s_2 \& r_1 \) in \( Q \) are compatible. The corresponding entries are shown with “c” in Table 1(b).

**Choice.** We assume that protocol trees have isomorphic choice branches. Nodes with type ▽ are only compatible with nodes with type ▼. The corresponding entries in the IAM are represented with “c” in Table 1. Figure 5(a)
Fig. 5: Possible Mismatches for Relations $\forall$ with $<$ and $\&$

presents that protocol $P$ has a choice $s_1 \forall s_2$, then $Q$ expects only one message executing either $r_1$ or $r_2$. Also, if a protocol has a choice $s_1 \forall r_2$, then the other part should have $s_2 \forall r_1$ and only one interaction is executed; see Figure 5(b).

Interactions having nodes with type $\forall$ are incompatible with other nodes with type $<$ or $\&$ and the corresponding entries in the IAM are denoted with “d” in Table 1. Figure 5(c) shows that $i_1$ and $i_2$ cannot be executed because they reach a deadlock. For example, $s_1 \forall s_2$ in $P$ means that only one node sends
Fig. 6: Possible Mismatches for Relations $<^*$ and $&^*$
a message; therefore, if \( Q \) expects a message in a sequence \( r_1 < r_2 \) or \( r_2 < r_1 \), then only one receiving node can be executed and the system deadlocks. This deadlock is repeated in symmetrical cases, for example, \( s_1 < s_2 \) or \( s_2 < s_1 \) in \( P \) and \( r_1 \nless r_2 \) in \( Q \); see Table 1(a).

In Figure 5(d), if \( s_1 \nless r_2 \) in \( P \) and \( s_2 < r_1 \) in \( Q \), then only one node can be executed in \( P \); therefore, the system deadlocks either if \( Q \) sends \( m_2 \) and it left waiting for a message \( m_1 \) that never is sent by \( P \), or if \( Q \) sends \( m_2 \) and it is never accepted by \( P \). This deadlock is also reached in symmetrical cases: \( s_1 \nless r_2 \) in \( P \) and \( r_1 < s_2 \) in \( Q \), \( s_2 \nless r_1 \) in \( P \) and \( s_1 < r_2 \) or \( r_2 < s_1 \) in \( Q \); see Table 1(b).

Figure 5(e) illustrates that \( P \) and \( Q \) deadlock since \( P \) sends either \( m_1 \) or \( m_2 \) (\( s_1 \nless s_2 \)) while \( Q \) expects both messages in parallel (\( r_1 \& r_2 \)). Also, this deadlock is reached in the symmetrical case if \( r_1 \nless r_2 \) in \( P \) and \( s_1 \& s_2 \) in \( Q \); see Table 1(a).

In Figure 5(f) we show that \( P \) and \( Q \) deadlock if \( s_1 \nless r_2 \) in \( P \) and \( s_2 \nless r_1 \) in \( Q \) because \( P \) either sends \( m_1 \) or expects \( m_2 \) but not both. This is valid for the symmetrical case \( s_1 \& r_2 \) in \( P \) and \( s_2 \nless r_1 \) in \( Q \); see Table 1(b).

**Loops.** The IAM described in Table 1 considers message nodes and behavioral relations with multiplicity 1; that is \( n^1 \) \& \( n^1 \) with \( \& \in \{<1,\&1, \nless1\} \). However, there are other cases we can compare: \( n^{1,*} \& n^{1,*} \) with \( \& \in \{<*,\&*, \nless*\} \).

If a control loop has one child, then the multiplicity is “pushed down” to the child, that is \( n^* \). Due to space limitations, we restrict ourselves to relations \( n^1 \& n^1 \) with \( \& \in \{<*,\&*, \nless*\} \) and the case \( n^* \& n^* \) is left as further work. Therefore, we abstract multiplicity 1 and \( * \) of relations \( <1, <*, \&1, \&*, \nless1, <*, \&*, \nless* \) to use the IAM in Table 1 for protocols containing loops with the same multiplicity; see Figure 6.

The following section describes how to construct a generic parallel adaptor from a minimal set of interactions needing adaptation.

### 4 Adaptor Construction Method

We define an automated method for constructing adaptors. The method consists of two steps. First, it finds the minimal set of interactions using the IAM. Next, it constructs a minimal adaptor from this set.

#### 4.1 Finding the Minimal Set of Interactions

We have detailed every entry of the IAM in Table 1. Now we explain how to use the matrix to identify the minimal set of interactions needing adaptation.

First, we need to build the set \( I \) of interactions between \( P \) and \( Q \). For example, in Figure 1 \( P \) and \( Q \) have nine interactions, indicated with dotted arrows; therefore, \( I \) contains nine ordered pairs.

Next, we look up the action specified for every ordered pair \( i_1=(w, x) \) and \( i_2=(y, z) \in I \) in the IAM, using the algorithm of Figure 7. Lines 5-6 match entries of Table 1(a). Symbol \( \oplus \in \{<,\&, \nless\} \) represents a behavioral relation between two nodes in the same protocol tree, as defined in Section 2.2. For example, for
procedure MatchingIAM(P, Q)
    I ← \{(x, y) | (x ∈ M_P ∧ y ∈ M_Q) ∨ (x ∈ M_Q ∧ y ∈ M_P) ∧ x = y\}
    for (w, x) ∈ I do
        for (y, z) ∈ I do
            if w ∈ M_P, y ∈ M_P or w ∈ M_Q, y ∈ M_Q then
                I_a ← I_a ∪ \{IAM[w ⊕ w, y][x ⊕ x, z]\}
            else if w ∈ M_P, z ∈ M_P or w ∈ M_Q, z ∈ M_P then
                I_a ← I_a ∪ \{IAM[w ⊕ w, z][y ⊕ y, x]\}
            else
                print "Deadlock"; exit 0
        end if
    end for
end procedure

Fig. 7: Algorithm to find the interactions needing to be adapted

protocols of Figure 1 if i_1 = (SendFlightSelected, RecFlightSelected) and i_2 = (SendClientID, RecClientID), then in both cases ⊕ = <. Next, line 6 looks up the interaction specified in IAM[SendFlightSelected < SendClientID][RecClientID < RecFlightSelected]. Therefore, i_1 has to be adapted and added to I_a.

Similarly, lines 7-8 match entries of Table 1(b). In lines 5 and 7, the algorithm considers symmetrical cases of the IAM.

The MatchingIAM algorithm outputs I_a, which is the minimal set of interactions needing to be adapted. If two interactions have incompatible control types or they wait for a message indefinitely, then the algorithm determines that no adaptor can be generated; see lines 9-10.

Using the MatchingIAM algorithm with the protocols of Figure 1 as input, we obtain: I_a = \{(SendFlightSelected, RecFlightSelected), (SendPaymentType, RecPaymentType), (SendInvoice, RecInvoice), (SendTicketInfo, RecTicketInfo)\}.

4.2 Constructing Minimal Adaptors

We want to construct a generic adaptor from the minimal set of interactions I_a, which is compatible with the protocols using synchronous communication.

In this adaptor, every interaction (s, r) ∈ I_a is put in a sequence with r before s. In principle, the sequences are put in parallel with each other to construct a generic adaptor. The sequences are ordered using AND and XOR nodes as follows. If some subset of message nodes that need to be adapted are part of a XOR branch in one of the protocol trees, then they should belong to a XOR branch in the adaptor too, to ensure that the adaptor terminates properly. Sequences inside a choice branch are put in parallel (AND), to be as least restrictive as possible.
To construct this adaptor, we first generate a skeleton tree from one of the protocol trees. The skeleton tree preserves each XOR node from the input tree, but control nodes in the input tree that are between two XOR nodes or between a XOR node and a leaf node, are abstracted into an AND node.

From the IAM follows that two protocol trees that can be adapted have isomorphic choice branches. Thus, the skeletons they yield are isomorphic too, so we can safely consider one arbitrary protocol tree to generate a skeleton. For example, the skeleton of protocol in Figure 3 is generated by abstracting the control nodes inside each XOR branch into an AND node, so the skeleton consists of a root XOR and two AND children.

To ensure this, we generate from a protocol tree a skeleton, which only contains parallel and choice branches. Next, we define a function to construct the adaptor given the set $I_a$ and the skeleton of $P$ (or $Q$).

Let $P$ be a protocol tree. A skeleton for $P$, written $\text{Skeleton}(P)$, is a tuple $(M, C, child, ctype, mult, root)$ where

- $M = \emptyset$
- $C = \{c \mid c \in C_p \land \text{ctype}_P(c) = \text{XOR}\} \cup \{x \mid x \in C_p \land \text{ctype}_P(c) = \text{XOR} \land (x, c) \in \text{child}_P\}$
- $\text{child} = \{(x, c) \mid (x, c) \in \text{child}_P \land \text{ctype}_P(c) = \text{XOR}\} \cup \{(x, c) \mid x \in C \land \text{ctype}_P(c) \neq \text{XOR} \land \text{there is no other } c' \in C \text{ such that } (x, c') \in \text{child}_P \land (c', c) \in \text{child}_P\}$
- $\text{ctype} = \{x \mapsto \text{XOR} \mid \text{ctype}_P(x) = \text{XOR}\} \cup \{x \mapsto \text{AND} \mid \text{ctype}_P(x) \neq \text{XOR}\}$
- $\text{mult} = \text{mult}_P[(C, 1, \ast)]$
- $\text{root} = \text{root}_P$

Using the skeleton $S$ constructed for a protocol $P$, we can construct the adaptor by inserting into $S$ a sequence $X$ of $r$ before $s$, where $(s, r) \in I_a$. The sequence $X$ is always child of an AND node. Each AND node in the skeleton has by construction a parent of type XOR, so the sequence belongs to a XOR branch. The sequence is child of an AND node if and only if that AND node abstracts either $s$ or $r$ in the skeleton generation procedure (depending on whether $s$ or $r$ is part of $P$).

Let $R$ be the skeleton of protocol $P$. Then the adaptor $A$ is a tuple $(M^+, M^-, C, child, ctype, mult, root)$ where

- $M^+ = \{x \mid \exists y : (x, y) \in I_a\}$
- $M^- = \{x \mid \exists x : (x, y) \in I_a\}$
- $C = C_R \cup \{(s, r) \mid (s, r) \in I_a\}$
- $M_A = M^+ \cup M^-$
- $\text{child} = \{(x, c) \mid (x, c) \in \text{child}_P \land \text{ctype}_P(c) = \text{XOR}\} \cup \{(x, c) \mid x \in C \land \text{ctype}_P(c) = \text{AND}\}$
- $\cup \{(x, c) \mid c = (s, r) \land (x = s \lor x = r)\}$
- $\cup \{(x, y) \mid x = (s, r) \land y \text{ is an AND node that abstracts } y \text{ in } P\}$
- $\text{ctype} = \{x \mapsto \text{XOR} \mid \text{ctype}_P(x) = \text{XOR}\} \cup \{x \mapsto \text{AND} \mid \text{ctype}_P(x) = \text{AND}\} \cup \{x_i \mapsto \text{SEQ} \mid i \in I_a\}$
Therefore, $A$ is the minimal protocol adaptor of $P$ and $Q$. If $M_A$ is empty, then $P$ and $Q$ are compatible and do not need an adaptor, and thus the adaptor is not generated. The minimal protocol adaptor $A$ constructed for the example in Figure 1 is depicted in Figure 2. This adaptor is compatible with protocols $P$ and $Q$ by construction. This minimal because it contains only those interactions needing to be adapted without adapting all interactions of $I$.

The minimal adaptor adapts $|I_a| = n_a$ interactions. Protocols $P$ and $Q$ have $|I| = n$ interactions between them. Every interaction has two messages nodes. If an adaptor contains all interactions, then it has $2n$ messages nodes. Therefore, the minimal adaptor reduces $(1 - 2n_a/2n)^*100\%$ of the number of messages exchanged by the composed services. For example, the minimal adaptor of Figure 2 reduces $56\%$ the overhead of messages exchanged at run-time.

Set $I$ can be constructed in quadratic time. To compare every pair of interactions, the algorithm MatchingIAM of Figure 7 has to look up $(n - 1)n/2$ entries in the IAM. Moreover, calculating the lca of every pair of nodes takes linear time. Also, the adaptor $A$ is constructed efficiently with the definition of protocol tree. Therefore, the performance of the complete method is quadratic for protocols containing parallelism.

5 Tool and Healthcare Example Case Study

In this section, we apply the two steps method to a teleradiology [10] example case. We have implemented the method in a tool which analyze at design-time two BPEL protocols. Next, this transforms protocols in trees and automatically identifies the minimal set of interactions $I_a$ or a mismatch that cannot be resolved. If the set is empty, then the tool does not generate any adaptor. Next, the tool constructs the minimal protocol adaptor using the skeleton structure and the method defined in Section 4.2.

The teleradiology process concerns the acquisition and interpretation of radiology scans; for example X-Ray scans. This process involves a hospital requesting the acquisition of radiology scans to a laboratory. The laboratory makes the interpretation of scans but this process can be also outsourced to a third-party.

The example case defined is a process variant of the teleradiology process defined in [10]. This case considers that the hospital always requests an urgent report to the laboratory before it sends a pre-diagnosis and a final report. The teleradiology process starts when the hospital request a scan, sending an order, a date and the patient data to the laboratory. Next, the hospital requests an urgent report and waits for the scans and the pre-diagnosis report. The laboratory sends the scans and the pre-diagnosis report. After that, the hospital requests an extra analysis to be included in the final report and it expects the urgent report. Finally, the laboratory sends the urgent report and final report to the hospital; see Figure 8.
Figure 9 presents the tool\textsuperscript{1} showing the protocol trees of the hospital (left-side), the laboratory (center) and the adaptor protocol tree (right-side). The set $I_a$ is depicted at the bottom of this figure. The tool generates the adaptor protocol tree in less than 1 second, using a desktop PC (2.3GHz Dual Core, 2GB RAM). Next, we can easily transform the adaptor tree into BPEL code to be executed in an engine for the demonstrator prototype. The minimal protocol adaptor for this case is depicted in Figure 10.

Figure 9 shows that protocols $P$ and $Q$ have $|I| = 12$ interactions, i.e., 12 message nodes each one. A protocol adapting 12 interactions has 24 message nodes. The tool found $|I_a| = 8$ interactions to be adapted, and the adaptor generated has 16 messages nodes. Using this adaptor, we reduce in 33\% the number of the messages exchanged at run-time.

\textsuperscript{1} Tool and paper examples are available at http://is.ieis.tue.nl/staff/rseguel/BpelAdaptor1.0.zip
Fig. 9: BPEL Adaptor prototype applied in the teleradiology example case

We ran this tool in the same desktop PC with other bigger examples: protocols containing sequences, parallelism, choices and loops (same multiplicity) with 16, 24, 36, 45 nodes; getting the adaptor in less than 1 second too.

6 Related Work

The idea of using minimal adaptors is due to Kumar and Shan [6]. They use heuristics to construct a minimal adaptor. However, the heuristics they use are difficult to understand, since they are based on the state space of the two protocols, and therefore difficult to interpret in terms of the syntax of the original protocols. The heuristics we present in this paper are all defined on the syntax of the protocols, and thus they are more easy to understand than those of Kumar.
Fig. 10: Adaptor for the teleradiology example case

and Shan. Moreover, a minimal adaptor generated by the Kumar/Shan approach may require that some send actions of the original protocols are blocked by the minimal adaptor, so the behavior of the original protocols is then restricted. Whereas the minimal adaptor we construct does not force send actions to be blocked, and thus it does not restrict the behavior of the original protocols in any way.

There are other related research on constructing protocol adaptors in the context of BPEL [3,7,11], based on earlier work by Yellin and Strom [12], but all these works focus on building adaptors that are not minimal, so each message is processed by the adaptor. Therefore, the adaptor is less efficient in terms of process complexity and the adaptation leads to more messages exchanged by the composed services at run-time.

Table 2 shows a comparison of this work with these related research, highlighting the contribution in the ‘IAM method’ column. Only this work and [6] generate a minimal adaptor. All related works assume synchronous communication; however, [3] and [6] also assume asynchronous communication. Every method is based on a formal model, but [6] in heuristics.

Like [6], we assume structured processes whereas the other approaches assume unstructured processes. Moreover, every adaptation method assumes processes with deterministic choices.

Next, [7,11,12] are only defined for sequential processes, while our method and [3,6] work with processes containing parallelism, choices and loops. Wang et al. [11] have defined a behavioral run-time adaptation method; however, all other works define a design-time behavioral adaptation method. A more detailed and complete overview can be found in [9].

7 Conclusions and Further Work

We have presented an efficient, automated method to construct a minimal adaptor for two business protocols containing parallelism and loops. The method consists of two steps. First, it finds the minimal set of interactions needing
## Table 2: Comparison of methods for generating protocol adaptors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptor type</td>
<td>Minimal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Comm. semantics</td>
<td>Synchronous</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Asynchronous</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Formal model</td>
<td>FSM</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Petri Nets</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Pseudocode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Protocol Trees</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Process type</td>
<td>Structured</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Non-structured</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Process choices</td>
<td>Deterministic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Non-deterministic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Process concurrency</td>
<td>Sequential</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Parallel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Choice</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Loop</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Time of computation</td>
<td>Design-time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Run-time</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

adaptation, analyzing the protocol syntax to identify behavioral mismatches. Key part is the algorithm that, given a set of interactions, identifies exactly those interactions needing adaptation without calculating the combined states of the parallel protocols.

Secondly, it builds in an efficient way a generic adaptor from the identified minimal set of interactions. The minimal adaptor has lower process complexity and is more efficient than adaptors that are constructed by current protocol adaptor generation approaches, since these adaptors process all interactions. This reduces the overhead of messages exchanged at run-time. The performance of the complete method is quadratic for protocols containing parallelism.

Therefore, this method makes service adaptation becomes more efficient, which is key for the Service Oriented Architecture (SOA) paradigm.

There are several directions for future work. We are currently extending the IAM method for asynchronous communication. We plan to extend the method for interface adaptation [8]. We will continue extending the prototype, considering links and loops with different multiplicity and applying it to more real-life case studies.

### References


