The Service Adaptation Machine

Kenneth Wang[1] Marlon Dumas[2,1] Chun Ouyang[1]
[1] *Faculty of IT, Queensland University of Technology, Australia*
{kw.wang,c.ouyang}@qut.edu.au
[2] *Institute of Computer Science, University of Tartu, Estonia*
marlon.dumas@ut.ee

Julien Vayssière
SAP Research Centre Brisbane, Australia
julien.vayssiere@sap.com

Abstract

The reuse of software services often requires the introduction of adapters. In the case of coarse-grained services, and especially services that engage in long-running conversations, these adapters must deal not only with mismatches at the level of individual interactions, but also across interdependent interactions. Existing techniques support the synthesis of adapters at design-time by comparing pairs of service interfaces. However, these techniques only work under certain restrictions. This paper explores a runtime approach to service interface adaptation. The paper proposes an adaptation machine that sits between pairs of services and manipulates the exchanged messages according to a repository of mapping rules. The paper formulates an operational semantics for the adaptation machine, including algorithms to compute rule firing sequences and criteria for detecting deadlocks and information loss. The adaptation machine has been implemented as a prototype and tested on common business processes.

Keywords: *Service-Oriented Architecture, Conversational Service, Adaptation*

1. Introduction

Service-oriented architectures decompose systems into coarse-grained software artifacts (namely services) that expose information, rules and processes across ownership domains. This approach leads to architectures that are aligned with the business domain and can seamlessly evolve in response to changes in business requirements. On the other hand, since services are coarse-grained, their reuse and evolution often require the introduction of complex adapters. Indeed, when services encapsulate long-running business processes, their interfaces are often conversational, meaning that they capture multiple interactions related through control-flow dependencies. Thus, when reusing a service, one needs to deal with mismatches not only at the level of individual interactions, but also across interactions occurring in the context of a conversation.

For example, Figure 1 shows the interfaces of two incompatible services (customer and supplier). To capture the behavioural aspects of these interfaces, we use the Business Process Modelling Notation (BPMN)[1]. The customer service sends a *PurchaseOrder* (*PO*) and then expects to receive a single *OrderResponse*. The supplier on the other hand receives a *PO* and replies with a *POResponse* followed by one or more *POUpdate*s (to keep the customer informed of the fulfillment of the order), until it finally issues a *POConfirm* to mark the completion of the fulfillment process. This example is not unrealistic. The interfaces in this example are taken from industry standards: the interface on the left corresponds to a fragment of an xCBL *order management choreography*[2] while the required interface corresponds to a RosettaNet *partner interface process*[3].

A common approach to reconcile incompatibilities such as those exposed by this example is to introduce adapters that resolve differences between pairs of services on a case-by-case basis. This approach lacks scalability and leads to a proliferation of adapters that need to be maintained as the underlying services and their interconnections evolve.

[1] `http://www.bpmn.org`
[2] `http://www.xcbl.org`
[3] `http://www.rosettanet.org`

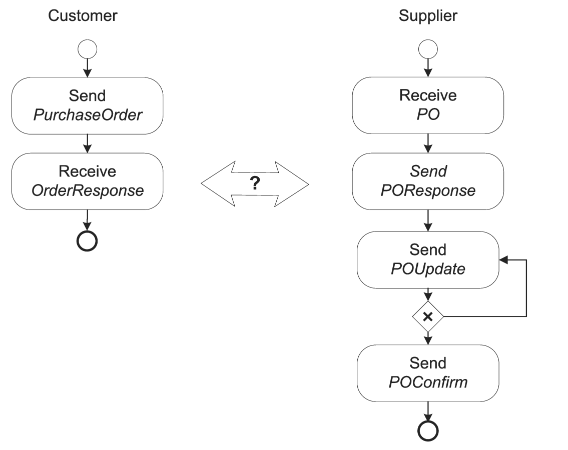

Figure 1. Incompatible conversational service pair.

As an alternative, this paper proposes a *service adaptation machine* that reconciles pairs of incompatible services by intercepting, transforming and forwarding messages according to a set of *mapping rules*. The adaptation machine masks and resolves incompatibilities by enacting the role of a compatible partner for each of the interacting services. The mapping rules have the form of production rules. Each rule describes a transformation between one or multiple source message types and a target message type. The adaptation machine fires these rules in a chained manner to produce messages expected by the interacting services based on previously intercepted messages. Importantly, the mapping rules are only fired if their output is required. By tracking the state of each interacting service (with respect to their behavioural interface) the adaptation machine is also capable of detecting two types of undesirable scenarios:

Deadlock: At least one of the interacting services expects a message but none of the services can produce any further messages.

Information loss: A message sent by a service is not consumed in any form by the intended recipient.

The paper provides a precise operational semantics of the adaptation machine, including backward-chaining algorithms for computing rule firing sequences and conditions for detecting deadlocks and information loss at runtime. The adaptation machine has been implemented as a prototype and tested on common business-to-business protocols.

It is not in the scope of this paper to determine whether or not a given set of mapping rules is sufficient to fully reconcile differences between a pair of services. Instead, the paper focuses on the problem of determining, at runtime, if differences can be resolved. Prior work has addressed the issue of statically determining if two given interfaces (or protocols) can be reconciled by means of an adapter. However, these techniques only work for restricted classes of behavioural interfaces and mapping rules (for a detailed discussion, see Section 6).

The rest of the paper is structured as follows. Section 2 introduces the notion of service interface used in this paper. Section 3 defines the syntax and informal semantics of mapping rules. Section 4 presents a formal operational semantics of the adaptation machine. Section 5 provides the proof-of-concept prototype. Section 6 discusses related work, and Section 7 concludes.

2. Behavioural Service Interfaces

The behavioural interface of a service captures control-flow dependencies over the set of possible message exchanges between the service and its environment. We represent a behavioural service interface as a Finite State Machine (FSM), that we call a *service interface FSM*. The choice of FSMs is motivated by the following reasons:

- It is arguably the simplest and most widely understood model of system behaviour and it has been used in several previous work in the area of behavioural service interface analysis [3, 4, 15].
- It is sufficiently powerful to capture most forms of behaviour encountered in service interfaces, including race conditions between messages and *interleaved* parallel execution.
- There exist transformations from other notations for service behaviour modelling to FSMs. In particular several transformations from BPEL to FSMs are implemented in existing tools such as WS-Engineer [11].

A service interface FSM consists of *states* and *transitions*. Each transition is labelled either by a *send* or by a *receive* action. Concretely, a transition's label includes a *direction* (! for send and ? for receive) and a message type. For example, we write !*PurchaseOrder* to denote the action of sending a purchase order. We refer to message types by names without describing their underlying structure. This is assumed to be captured in the structural interface which may be represented for example using the Web Service Description Language (WSDL) [8]. Also, we represent only one send or receive action per transition. Often, a send and receive actions come in pairs (i.e. a request and a response). While in WSDL these two message exchanges would be grouped into a single *operation*, in our model we represent them separately. This allows us to keep our model simple without loosing generality.

A service interface FSM contains exactly one start state (i.e. a state without incoming transitions), at least one end state (i.e. a state without outgoing transitions), and at least one transition (i.e. a service must send or receive at least one message). A service interface FSM captures the behaviour of each *instance* of a service, from the start state all the way to one of the end states. When a service instance reaches any end state, it is said to have completed successfully.

A service interface FSM is a quintuple $FSM = (S, T, s_0, F, \delta)$, where:

- S is a set of states.
- $T = !T \cup ?T$ is a set of actions, where
  - $!T$ is a set of send actions, and
  - $?T$ is a set of receive actions.
- $s_0 \in S$ is the start state.
- $F \subset S$ is a set of end states.
- $\delta : S \times T \rightarrow S$ is a state transition function.

For example, Figure 2 depicts respectively the FSMs of the Customer and the Supplier interfaces in Figure 1. The FSM of the Customer interface is written $FSM_c = (S_c, T_c, s_0^c, F_c, \delta_c)$ where:

- $S_c = \{1, 2, 3\}$
- $T_c = !T_c \cup ?T_c$ where
  - $!T_c = \{!PurchaseOrder\}$, and
  - $?T_c = \{?OrderResponse\}$.
- $s_0^c = 1$
- $F_c = \{3\}$
- $\delta_c = \{(1, !PurchaseOrder) \rightarrow 2,$
  $(2, ?OrderResponse) \rightarrow 3\}$

The FSM of the Supplier interface is written $FSM_s = (S_s, T_s, s_0^s, F_s, \delta_s)$ where:

- $S_s = \{1, 2, 3, 4, 5\}$
- $T_s = !T_s \cup ?T_s$ where
  - $!T_s = \{!POResponse, !POUpdate,$
    $!POConfirm\}$, and
  - $?T_s = \{?PO\}$.
- $s_0^s = 1$
- $F_s = \{5\}$
- $\delta_s = \{(1, ?PO) \rightarrow 2,$
  $(2, !POResponse) \rightarrow 3,$
  $(3, !POUpdate) \rightarrow 4,$
  $(4, !POUpdate) \rightarrow 4,$
  $(4, !POConfirm) \rightarrow 5\}$

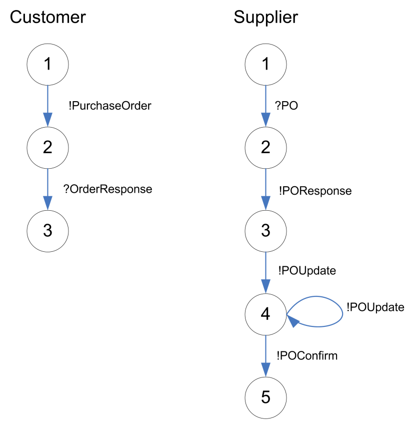

Figure 2. FSMs for the Customer and the Supplier interfaces in Figure 1.

3. Mapping Rules

A mapping rule $I \rightarrow^{tx} o$ is composed of one or multiple input patterns $i_1 \ldots i_n$ $(n \geq 1)$, one output template o, and a data transformation function tx. The transformation function describes how the messages that match the input patterns are converted to the desired output. For example, if messages were represented in XML, one could encode the transformation functions in XSLT. But other languages could be used for this purpose and this choice is orthogonal to the operation of the adaptation machine. The definition of data transformation functions is outside the scope of this paper. Accordingly, we will often omit the transformation functions when representing mapping rules.

3.1. Syntax of Rules

Input patterns may take different forms, which corresponds to a message type. In its simplest form, an input pattern is just a message type. Such pattern matches and consumes a single message of that type. Similarly, an output template can take the form of a message type, in which case the rule in question produces a single message of that type.

In some cases, we need input patterns that match multiple messages of the same type at once. For example, consider a scenario where a number of messages corresponding to purchase order items must be concatenated into a single message. To capture such scenario, we introduce the notion of *unbounded input pattern*. A pattern of this type takes the form $T1^{+}(T2)$ where $T1$ is the type of messages that are consumed by the pattern, and $T2$ is a *milestone*. This milestone is used to inhibit the pattern from matching a set of messages of type $T1$, until a message of type $T2$ is also available for consumption. The milestone is necessary because the number of messages to be consumed is not known *a priori*. For example, a rule $POItem^{+}(POConfirm) \rightarrow OrderItems$ consumes one or multiple messages of type *POItem* provided that a message of type *POConfirm* has been received. The rule then collapses all the matched messages of type *POItem* into a single message of type *OrderItems*. Importantly, the rule firing does not consume any message of milestone type *POConfirm*, even though it requires at least one message of this type to be available.

The $^{+}$ symbol used to define unbounded input patterns, can also be used to define output templates. However, in this case, the milestone is not needed: instead the rule will produce as many messages as the transformation function yields. For example, if a message of type *OrderItems* contains an array of ten items, then a mapping rule $OrderItems \rightarrow POItem^{+}$ will produce ten messages of *POItem* from one message of type *OrderItems*.

The following EBNF describes the syntax of mapping rules:

```
Rule ::= RuleInputs → RuleOutput
RuleInputs ::= InputItem {InputItem}
RuleOutput ::= OutputItem
InputItem ::= MessageType | UnboundedInputType
OutputItem ::= MessageType | UnboundedOutputType
UnboundedInputType ::=
                MessageType+(MilestoneMessageType)
MilestoneMessageType ::= MessageType
UnboundedOutputType ::= MessageType+
```

Mapping rules must also abide to an additional syntactic restriction. The same message type may not appear twice in the input pattern of a mapping rule. Formally: $\forall i_1, i_2 \in I : i_1 \neq i_2$.

As an example, Table 1 shows a mapping rules repository *MR*. *MR* contains 10 mapping rules which can be used for mediating between the two service interfaces shown in Figure 2. A repository is a collection of rules accumulated over a number of adaptation scenarios. Thus, not every rule in a rule repository is relevant to every scenario. In Section 4, which presents the operational semantics of the adaptation machine, we will discuss how rules are selected and used in a given adaptation scenario.

id	rule syntax
mr_1	$PurchaseOrder \rightarrow POrder$
mr_2	$POrder \rightarrow PO$
mr_3	$POrder, OrderDetails \rightarrow PO$
mr_4	$POResponse, OrderUpdate \rightarrow OrderResponse$
mr_5	$POUpdate^{+}(POConfirm) \rightarrow OrderUpdate$
mr_6	$POItems \rightarrow OrderItems$
mr_7	$POConfirm \rightarrow OrderConfirm$
mr_8	$POConfirm \rightarrow OrderUpdate$
mr_9	$PurchaseOrder \rightarrow OrderDetails$
mr_{10}	$POItems \rightarrow OrderItem^{+}$

Table 1. A list of mapping rules in a repository.

3.2. Firing of Rules

To fire a mapping rule, one or multiple messages of the message types that match the rule's input patterns need to be available. After the rule is fired, a message with the message type that fit into the rule's output template will be produced. Sometimes it is possible that, given the available messages, none of the rules can be fired to directly produce the message(s) of the required type. In this case, it may be necessary to fire more than one rule to complete the transformation. A *firing sequence* therefore refers to an ordered list of mapping rules where firing of these rules in sequence can lead to the production of the required message(s). For example, consider the set of mapping rules *MR* in Table 1. Given a message of type *PurchaseOrder*, a message of type *PO* can only be produced by firing a sequence of rules $[mr_1, mr_2]$ in *MR*.

4. Operational Semantics

In this section, we describe the behaviour of an adapter in terms of initialisation, adaptation cycle, and termination. An adapter *A* between two service interfaces *IA* and *IB* is hereby defined as a triplet (FSM_{IA}, FSM_{IB}, *MR*) where:

- FSM_{IA} is the FSM of the service interface *IA*.
- FSM_{IB} is the FSM of the service interface *IB*.
- *MR* is the set of mapping rules (i.e. mapping rules repository).

4.1. Initial State

An adapter is instantiated and executed for the purpose of mediating between a pair of service instances. Messages exchanged between the pair of instances are thus routed to an instance of the corresponding adapter by means of a correlation mechanism from which we abstract in this paper. Several such correlation mechanisms exist, including for example the correlation mechanism supported by BPEL based on message properties defined in terms of XPath expressions.

We define the execution state of an adapter (instance) A as a hextuple A_S = ($state_{IA}$, $state_{IB}$, B^R, B^C, F, H), where:

- $state_{IA}$ and $state_{IB}$ are the current states of the service interfaces IA and IB respectively. Initially, the state of a service interface is the start state (i.e. s_0) of the respective FSM. Updates to the state ($state'$) can be derived from the state transition function δ using the current state ($state$) and an enabled action $t \in T$, i.e. $state' = \delta(state, t)$.

- $B = B^R \cup B^C$ is a set of **messages** within a message buffer. B^R denotes a set of messages that has been intercepted by the adapter. B^C denotes a set of messages that has been created by the adapter. Both B^R and B^C are empty upon initialisation.

- $F \subseteq B$ is a set of **messages** that have been forwarded by the adapter. The set F is empty upon initialisation.

- H records the history of rules firing. It is written as a set of pairs (msg, mr), where msg is a **message** and mr a **mapping rule**. A pair (msg, mr) denotes that the rule mr has fired, taking as input the message msg. The set H is empty upon initialisation.

In addition, to facilitate the definition of adaptation algorithm (in the next subsection), we introduce the following two auxiliary functions:

- Function E takes an adapter state and returns a set of **message types** that is currently *expected to be received* by the adapter. Let msgType be a function which takes a send or receive action and returns the message type being sent or received respectively. Hence, given an adapter state A_S, E can be derived from a set of send actions enabled at $state_{IA}$ for service interface IA and those at $state_{IB}$ for service interface IB, i.e. $E(A_S) = \{msgType(t) \mid t \in !T_{IA} \wedge (state_{IA}, t) \in dom^4(\delta_{IA})\} \cup \{msgType(t) \mid t \in !T_{IB} \wedge (state_{IB}, t) \in dom(\delta_{IB})\}$. If E is empty after initialisation (i.e. the adapter is not expecting any messages), then adaptation is not possible.

- Function R takes an adapter state and returns a set of **message types** that is currently *required to be forwarded* by the adapter. Given an adapter state A_S, R can be derived from a set of receive actions enabled at $state_{IA}$ and those at $state_{IB}$, i.e. $R(A_S) = \{msgType(t) \mid t \in ?T_{IA} \wedge (state_{IA}, t) \in dom(\delta_{IA})\} \cup \{msgType(t) \mid t \in ?T_{IB} \wedge (state_{IB}, t) \in dom(\delta_{IB})\}$. The set R may be empty upon initialisation (i.e. no message is required to be forwarded).

[4] *dom* is the domain of the function δ.

For example, the state of an adapter A between the interfaces in Figure 2 can be written $A_S = (state_c, state_s, B^R, B^C, F, H)$. Let A_{S_0} be the initial state of A, $A_{S_0} = (1, 1, \{\}, \{\}, \{\}, \{\})$, $E(A_{S_0}) = \{PurchaseOrder\}$, and $R(A_{S_0}) = \{PO\}$.

4.2. Adaptation Cycle

After initialisation, an adapter listens to the incoming messages from any of the mediated services. If an incoming message msg has a message type expected by adapter A at the initial state A_{S_0}, i.e. $msgType(msg) \in E(A_{S_0})$, the adapter triggers a cycle of internal actions to create the message that is required to be forwarded. These actions include rules selection, rules firing, message release, etc, and can be performed without interference from the external environment. We consider this series of actions constitute an *adaptation cycle*. The adapter performs as much as it can and subsequently returns to a "listening" mode until the next adaptation cycle commences or the terminating condition is reached.

Figure 3 depicts an overview of an adaptation cycle. When a new message is intercepted, the state of the service that sent this message is updated, and the message is added to the buffer. If the type of this message is expected by the target service, the message is forwarded. Each "expected" message type is checked in turn and the adaptation machine seeks to find a firing sequence to produce a message of that type. If such a firing sequence is found, it is executed and consequently some messages may be released. This process of finding and executing chains of mapping rules to produce expected messages is repeated until no more such rules can be found. At this point, the adaptation cycle completes. The shaded boxes indicate sub-routines of the adaptation cycle for which we provide algorithms later.

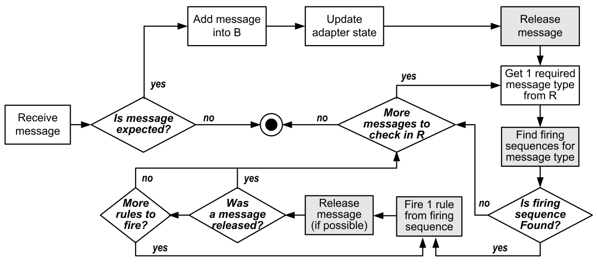
Receive message
Is message expected?
yes
no
Add message into B
Update adapter state
Release message
Get 1 required message type from R
yes
More messages to check in R
no
Find firing sequences for message type
Is firing sequence Found?
no
yes
Fire 1 rule from firing sequence
Release message (if possible)
Was a message released?
yes
no
More rules to fire?
yes

Figure 3. Flowchart of an adaptation cycle.

Algorithm 1 defines a function *adaptCycle* that captures more precisely the notion of adaptation cycle. The function takes as input an adapter A, a state A_S of adapter A, and an incoming message *newMsg* to A. It produces as output an updated adapter state after completion of the adaptation cycle. The function assumes as a precondition that the incoming message has a message type expected in state A_S. First, the new message is added to buffer B^R (line 3), and the states of both service interfaces *IA* and *IB* are updated respectively (lines 4-8). At this point, we check if any message can be released by calling a function *releaseMessages* (defined later) which returns a new adapter state (line 9).

Algorithm 1 Adaptation Cycle

```
1: FUNCTION adaptCycle(A: Adapter, A_S: AdapterState, newMsg: Message) : AdapterState
2: PRECONDITION $(msgType(newMsg) \in E(A_S))$
3: $A_S.B^R := A_S.B^R \cup \{newMsg\}$
4: if $(msgType(newMsg) \in A.FSM_{IA}.!T)$ then
5:     $state_{IA} := A.FSM_{IA}.\delta(A_S.state_{IA}, !msgType(newMsg))$
6: else
7:     $state_{IB} := A.FSM_{IB}.\delta(A_S.state_{IB}, !msgType(newMsg))$
8: end if
9: $A_S := releaseMessages(A, A_S)$
10: repeat
11:     $i := 1$
12:     $reqMessages := R(A_S)$
13:     repeat
14:         $rulesToFire := findFiringSeq(A, A_S, reqMessages[i], \{\})$
15:         $i := i + 1$
16:     until $(rulesToFire \neq []$ or $i > |reqMessages|)$
17:     if $(rulesToFire \neq [])$ then
18:         $i := 1$
19:         repeat
20:             $A'_S := A_S$
21:             $A_S := fireRule(A, A_S, rulesToFire[i])$
22:             $A_S := releaseMessages(A, A_S)$
23:             $i := i + 1$
24:         until $(A'_S.F \neq A_S.F)$
25:     end if
26: until $(rulesToFire = [])$
27: return $A_S$
28: END FUNCTION
```

Next, function *adaptCycle* starts to discover one or more firing sequences to generate a message type specified in $R(A_S)$ (line 10 and onwards). Given the current state A_S, *adaptCycle* traverses the list of message types in $R(A_S)$ to find if a firing sequence exists for generating a message type ($reqMessages[i]$) specified in the list (lines 13-16). The search for such a firing sequence is performed by calling a function named *findFiringSeq* (defined in Algorithm 2). The traversing of the list of message types in $R(A_S)$ stops as soon as *one* firing sequence is found (i.e. *findFiringSeq* returns a non-empty list of mapping rules) or otherwise until the entire list is traversed. Once a firing sequence is found, it will fire (lines 17-25). By calling the function *fireRule* (defined in Algorithm 3), the list of mapping rules in the firing sequence are fired in order one at a time (line 21). After each rule firing, it is checked *again* that if any message can be released (line 22). Since *findFiringSeq* and *releaseMessages* would update the state of the adapter, and in turn, the list of message types in $R(A_S)$, another firing sequence is then sought after. The search continues until *no* firing sequence can be found (i.e. *findFiringSeq* returns an empty list of mapping rules) after traversing the entire list of message types in $R(A_S)$ (ending at line 26). An adaptation cycle is completed when no more internal actions are possible. At this point, the adapter returns to listen to incoming messages. Once a message arrives satisfying the precondition for executing *adaptCyle*, another adaptation cycle will start. This process is repeated until a terminating condition is fulfilled, i.e. both service interfaces reach their end states.

4.3. Rules Selection

Algorithm 2 describes the function *findFiringSeq* used for rules selection within the adaptation cycle. This is a backward-chaining depth-first search algorithm which determines if there exist in the mapping rules repository *MR*, a set of rules for producing a certain message type. The function takes as input an adapter *A*, an adapter state A_S, a required message type *msgT*, and a set of message types that have been *visited* along the search. It returns a list of mapping rules that, if not empty, will at least produce *msgT* when fired.

Algorithm 2 Rules Selection

```
1: FUNCTION findFiringSeq(A: Adapter, A_S: AdapterState, msgT: MessageType,
   visited: Set <MessageType>) : List <Rule>
2: $visited := visited \cup \{msgT\}$
3: $rules := \{rule \mid rule \in A.MR \wedge o_{rule} = msgT\}$
4: for ($l$ = 1 to $|rules|$) do
5:     count := 0
6:     $rulesToReturn := []$
7:     $rule := rules[l]$
8:     for ($j$ = 1 to $|I_{rule}|$ where $I_{rule}[j] \notin visited$) do
9:         $i_{rule} := I_{rule}[j]$
10:        if $(existMatch(A_S.B, rule, i_{rule}))$ then
11:            count := count + 1
12:        else
13:            $tempRules := findFiringSeq(A, A_S, i_{rule}, visited)$
14:            if $(tempRules \neq [])$ then
15:                $rulesToReturn := tempRules \oplus rulesToReturn$
16:                count := count + 1
17:            end if
18:        end if
19:        $j := j + 1$
20:    end for
21:    if $(count = |I_{rule}|)$ then
22:        return $rulesToReturn \oplus [rule]$
23:    end if
24:    $l := l + 1$
25: end for
26: return []
27: END FUNCTION
```

The function starts by traversing all the rules in *MR* that can produce the output *msgT* (line 3). A rule in *MR* can be used if all the inputs of the rule are *available*. For each input of the rule, its availability can be checked via the auxiliary boolean function *existMatch* (line 10). Given a buffer B, a mapping *rule* and one of its input message type i_{rule}, *existMatch* returns $true$ if i_{rule} can be satisfied by the messages in B and has not been used by *rule* (i.e. $(i_{rule}, rule) \notin A_S.H$). Otherwise, it returns $false$. When *existMatch* returns $true$, the availability check moves onto the next input of *rule*. If *existMatch* returns $false$, function *findFiringSeq* is called recursively to determine if message type i_{rule} can be produced by another rule (line 13). For example, refer to rule $mr_5 : POUpdate^+(POConfirm) \rightarrow OrderUpdate$ in Table 1. Function *existMatch* returns $true$ for each of the inputs of mr_5, if there exist one or more *POUpdate*s (which has not been used by mr_5) and at least one *POConfirm* in the buffer B. The list of rules obtained from the recursive call(s) are added to *rulesToReturn* via list concatenation operator $\oplus$ (line 15).

Once all the inputs of a *rule* pass the availability check, this *rule* is added to *rulesToReturn* being collected along the check, and the search is completed (lines 21-23). Otherwise, the algorithm moves onto the next *rule* that can produce $msgT$, and repeat the above availability check. If *rulesToReturn* is empty after traversing the entire list of *rules*, there exists no firing sequence that can produce the required message type. Note that it is possible to fire alternatively *more than one* firing sequences (FRs) to obtain a required message type. The function *findFiringSeq* always returns the first one ($fr \in FRs$) that has been found, and the finding of fr among FRs is non-deterministic.

4.4. Rules Firing

Algorithm 3 specifies the firing of a rule in terms of the function *fireRule*. For each firing *rule*, the inputs of *rule* are collected into *ruleInputs* (lines 2-5). The *ruleInputs* are sought from the message buffer $B = B^R \cup B^C$. We introduce another auxiliary function *findMatch* (similar to *existMatch*) which takes as input the buffer B, a given *rule* and one of its input message type i_{rule}, and returns a set of messages in B that satisfy i_{rule}. This set of messages will be consumed when *rule* is fired. For a given input $i_{rule} \in I_{rule}$, if there exist more than one message of type i_{rule} (i.e. $\{msg_1, msg_2\} \subseteq B \wedge msgType(msg_1) = i_{rule} \wedge msgType(msg_2) = i_{rule}$), then it is non-deterministic whether msg_1 or msg_2 will be used. A possible solution is to select inputs based on FIFO (first in first out) strategy where messages arriving first to the adapter are always consumed first. Next, the history of rules firing is updated by adding the pairs of *rule* and message *msg* that is consumed upon the firing of *rule* (line 7). The transformation function tx_{rule} for firing *rule*, takes as input one or more messages in *ruleInputs* (of type I_{rule}) and produces one and only one output message *newMsg* (of type o_{rule}) (line 8). This output *newMsg* from the firing of *rule*, as a new message created by the adapter A, is added to the buffer B^C (line 9). Finally, function *fireRule* returns the updated adapter state A_S comprising the above information.

Algorithm 3 Rules Firing

```
1: FUNCTION $fireRule(A$: Adapter, $A_S$: AdapterState, $rule$: Rule)
   : AdapterState
2: $ruleInputs := \{\}$
3: for ($j = 1$ to $|I_{rule}|$) do
4:     $ruleInputs := ruleInputs \cup findMatch(A_S.B, rule, i_{rule}[j])$
5:     $j = j + 1$
6: end for
7: $A_S.H := A_S.H \cup \{(msg, rule) | msg \in ruleInputs\}$
8: $newMsg := tx_{rule}(ruleInputs)$
9: $A_S.B^C := A_S.B^C \cup \{newMsg\}$
10: return $A_S$
11: END FUNCTION
```

4.5. Message Release

In Algorithm 4, function *releaseMessages* can be executed to release any message *msg*, which is available in the buffer B but not in F and has a type specified in R as required to be forwarded. Upon message release, *msg* is added to F, the set of messages the adapter has forwarded (line 4). Also, the states of each of the two service interfaces are updated respectively (lines 5-9).

Algorithm 4 Release Messages

```
1: FUNCTION $releaseMessages(A$:Adapter, $A_S$:AdapterState) :
   AdapterState
2: while ($\exists msg \in (A_S.B \setminus A_S.F) : msgType(msg) \in R(A_S)$) do
3:     release $msg$
4:     $A_S.F := A_S.F \cup \{msg\}$
5:     if ($msgType(msg) \in A.FSM_{IA}.!T$) then
6:         $A_S.state_{IA} := A.FSM_{IA}.\delta(A_S.state_{IA}, !msgType(newMsg))$
7:     else
8:         $A_S.state_{IB} := A.FSM_{IB}.\delta(A_S.state_{IB}, !msgType(newMsg))$
9:     end if
10: end while
11: return $A_S$
12: END FUNCTION
```

4.6. Termination

An adapter enters an adaptation cycle when a message of an expected type arrives. We define a terminating condition for an adapter such that whenever the adapter is in a “listening” mode, and there is no expected message (i.e. $E = \{\}$), it terminates.

Ideally, we want an adapter to terminate when all the messages exchanged between the mediated services are fully utilised, and forwarded to their respective receiving services. This means a proper termination strategy requires the adapter to forward all the required messages (i.e. R = {}).

However, we identify a number of termination anomalies that can be derived from the set of variables given to an adapter instance. *Deadlock* and *Message Loss* are two of these anomalies.

Determining Deadlock A deadlock occurs when the adapter terminates, but a service is left waiting indefinitely for one or more messages to arrive. Deadlock scenario can be determined by checking if the adapter is still required to forward any messages when it terminates (i.e. $E = \{\} \wedge R \neq \{\}$).

Determining Message Loss Message loss occurs when a message is intercepted or created by an adapter, and at the end of the lifecycle of the adapter, is not used by any rules, or released by the adapter. This indicates that some information was conveyed in a message that did not find its way to a recipient service (e.g. a non-forwarded invoice). The adaptation machine can detect message loss by testing the following condition: $\exists msg \in B : (msg \notin F) \wedge (\forall mr \in MR : (msg, mr) \notin H)$

4.7. An Illustrative Example

We continue using the example of an adapter A mediating between the two service interfaces in Figure 2. After initialisation, the adapter A triggers the first adaptation cycle once a message *PurchaseOrder* arrives. The adapter state changes to $A_S = (2, 1, \{PurchaseOrder_1\}, \{\}, \{\}, \{\})$. Also, $E(A_S) = \{\}$ and $R(A_S) = \{PO, OrderResponse\}$. The adaptation cycle will attempt to create both *PO* and *OrderResponse* using the rules in Table 1 (i.e. *MR*) and the messages we have on hand (i.e. *B*).

We start with *PO*. When the function *findFiringSeq* is called, the first mapping rule that can produce *PO* (i.e. mr_2) responds. However, the inputs for mr_2 are not available in B. Hence, the function recursively calls itself with the required message type I_{mr_2} (i.e. *POrder*), and mr_1 responds this time. As the input for mr_1 is available in the buffer (i.e. $PurchaseOrder \in B$), this results in a first firing sequence $rulesToFire = [mr_1, mr_2]$.

When the sequence *rulesToFire* is fired, it produces two messages (to be added to B). One of these messages corresponds to *PO* and is released when the function *releaseMessage* is called. The state of the adapter becomes A_S = (2, 2, $\{PurchaseOrder_1\}$, $\{POrder_2$, $PO_3\}$, $\{PO_3\}$, $\{(PurchaseOrder_1$, $mr_1)$, $(POrder_2$, $mr_2)\}$). $E(A_S) = \{POResponse\}$ and $R(A_S) = \{OrderResponse\}$.

Next, a message *POResponse* will arrive (from the supplier interface). The adaptation will continue until both the customer interface and the supplier interface reach their final state (see FSMs in Figure 2) and $E(A_S)$ becomes an empty set. According to the definitions of deadlock and message loss in Section 4.6, it can be determined that the entire adaptation cycle is free from these two anomalies.

5. Tool Support

To materialise the concept introduced in this paper, we have embodied the above ideas in a tool called the *Service Mediation Engine* (Megine). Megine allows developers to register pairs of Web services whose interfaces are captured as a combination of WSDL definitions and FSMs. We argue that, while other languages such as BPMN and BPEL can be used to describe behavioural service interfaces at the design level, it is possible to translate these models to state machines for adaptation. Megine also manages a repository of mapping rules. Each rule refers to one or multiple XSLT transformations. Messages intercepted by the engine are assumed to be encoded in SOAP.

At any point in time, the mediation engine manages a number of adapters (as defined in the previous section). Each adapter reconciles differences between a specific pair of service instances. When a message is intercepted, it is first associated to an adapter before being processed. To this end, Megine assumes that every message contains an identifier (cf. the WS-Addressing *messageID* header) and (optionally) a reference to a previous related message (cf. the WS-Addressing *relatesTo* header). As an alternative to WS-Addressing, we could have used BPEL correlation sets, which provide a more general mechanism for message correlation that does not require specific headers to be included in all messages. We leave this as a possible future extension of the engine.

The mediation engine includes an administration console to monitor the state of the adapters and to view histories of intercepted, transformed and forwarded messages, and rule firings. It also includes functions for maintaining mapping rules. We have tested the mediation engine against a number of xCBL and RosettaNet choreographies such as those presented in Section 1.

6. Related Work

Mismatches between service interfaces can be classified into data (or schema) mismatches and behavioural (or protocol) mismatches. Commercial products such as SAP XI and Microsoft BizTalk incorporate graphical editors for specifying alignments between schema pairs from which executable transformations (e.g. in XSLT) are generated. However this approach requires significant human intervention. Many techniques have been devised to automate this schema matching process [16]. Some techniques exploit similarities between schema element names. Others map the schema elements to a conceptual domain represented as an ontology and exploit ontological relationships to resolve mismatches [14]. Our proposal is complementary to these techniques. It focuses on applying data transformations at the right time and in the right order.

There is also a significant body of work related to behavioural adaptation. These approaches adopt different models of concurrency to capture behavioural interfaces (i.e. protocols) such as labelled transition systems [12], Abstract State Machines (ASMs) [2,9], finite state automata [15, 18] or process algebra [5].

Yellin & Strom [18] define a notion of compatibility of components whose behavioural interfaces (protocols) are described as FSMs. They address the question of verifying if a given adapter (also specified as a FSM) is able to reconcile two incompatible protocols. They also discuss how to generate an adapter from links between *parameters* in the provided interface and corresponding parameters in the required interface. But there is an assumption that these adapters are not able to collapse multiple messages of the same type into a single message. Also, Yellin & Strom assume that each parameter in one interface is directly mapped to one or multiple parameters in the other interface through a single "mapping rule", but do not consider the possibility of chaining multiple mapping rules. Our approach overcomes these limitations. On the other hand, our approach computes the adapter at runtime rather than statically.

Another technique for static generation of protocol adapters is exposed in [17]. As in [18], the authors deal with one-to-one, one-to-many and many-to-one mismatches. However, they too do not address cases where an unbounded number of messages of one type must be collapsed into a single message. Similarly, [15] extend Yellin & Strom's work by allowing deadlocks to be pinpointed and by suggesting ways to resolve these deadlocks. This work also differs from ours in that it does not consider the use of chained mapping rules to resolve behavioural interface mismatches. Also Brogi et al. [6] propose a method for automatic generation of BPEL adapters for reconciling behavioural differences between two interacting BPEL processes. The main idea of their approach is to first identify an appropriate mirror behaviour for each of interacting BPEL process and then merge them into an adapter by connecting corresponding send and receive actions. Again, the authors do not address scenarios where multiple messages of the same type need to be collapsed into a single one, or vice-versa.

In [3], the authors identify a set of mismatch patterns between behavioural interfaces and provide templates of BPEL code that developers may reuse to build adapters that resolve these mismatches. However, the compositionality of these BPEL templates is not considered and thus the approach is not systematic. Similar mismatch patterns are identified in [9] where a high-level architecture for dealing with such mismatches is proposed. The ADAPT framework [1] goes further by proposing a notation for N-to-M mappings, i.e. mappings where data from N services are collected and repartitioned to M services. This is similar to mapping rules where multiple messages (each of a different type) are merged into one. However, in the ADAPT framework, messages produced by the repartitioning rules are forwarded to the target services eagerly without checking if the target service is in a state where it can consume these messages.

Altenhofen et al. [2] propose a formal model for protocol mediation based on ASMs. They propose a base ASM model for mediators and show that this model can be refined to deal with the mismatch patterns identified in [9].

The above approaches are based on static compatibility analysis and/or adapter synthesis. They have the advantage of being able to detect and resolve incompatibilities prior to deployment, but they also have limitations stemming from the inherent computational complexity of static analysis and synthesis. In contrast, our approach is based on runtime chaining and firing of mapping rules.

There is a resemblance between the mapping rules used by the adaptation machine and composite Event-Condition-Action (ECA) rules in active databases [7] or event-driven rules in Complex Event Processing (CEP) systems [13]. However, the firing semantics of the adaptation machine is different from those that have been considered in the database or CEP settings. The way rules are fired in the adaptation machine is lazy: a rule is only fired if it is part of a firing sequence leading to an output message that is expected by one of the interacting services. This firing style is akin to backward chaining in rule-based inference engines. On the other hand, the consumption semantics of the adaptation machine is different from those commonly considered in inference engines because a message can not be used to fire twice the same mapping rule, but at the the same time the same message can be used to fire different rules (i.e. to break down a message into several ones).

In previous work [10], we defined an algebraic approach to specify service adapters between pairs of interfaces. However, in this earlier work, the adapter construction is done manually using a predefined set of operators. In comparison, the adaptation machine presented here is able to implicitly infer a service adapter at runtime based on reusable mapping rules.

7. Conclusion and Future Work

This paper introduced the concept of a service adaptation machine that acts as an adapter between incompatible services, by selecting and chaining mapping rules to resolve mismatches as they arise. In addition to resolving mismatches, the adaptation machine detects deadlocks and possible information loss by comparing the messages it has buffered with the current state of the mediated services. We have defined an operational semantics of the adaptation machine, including algorithms to move the machine from a stable state to another when a new message arrives. These algorithms have been implemented in a prototype.

In future work, we plan to design techniques to statically detect whether or not a given set of mapping rules are sufficient to reconcile all differences between two service interfaces, and if not, to suggest additional mapping rules that may be introduced to achieve completeness of the rules.

Acknowledgment. This work was funded by an ARC Linkage Project (LP0455394) and SAP Research.

References

[1] G. Alonso, C. Pautasso, and B. Biörnstad. CS Adaptability Container. Deliverable #11, EU FP5 Project "ADAPT", August 2004.

[2] M. Altenhofen, E. Börger, and J. Lemcke. A High-Level Specification for Virtual Providers. *International Journal of Business Process Integration and Management*, 1(4):267–278, 2006.

[3] B. Benatallah, F. Casati, D. Grigori, H. Motahari Nezhad, and F. Toumani. Developing Adapters for Web Services Integration. In *Proceedings of the 17th International Conference on Advanced Information System Engineering, CAiSE 2005, Porto, Portugal*, pages 415–429, 2005.

[4] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *Int. J. Cooperative Inf. Syst.*, 14(4):333–376, 2005.

[5] A. Brogi, C. Canal, and E. Pimentel. On the Specification of Software Adaptation. *Electronic Notes in Theoretical Computer Science*, 97:47–65, 2004.

[6] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC 2006), Chicago, IL, USA*, pages 27–39, December 2006.

[7] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, pages 606–617, Santiago, Chile, September 1994. Morgan Kaufmann.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. `http://www.w3.org/TR/wsdl`.

[9] E. Cimpian and A. Mocan. WSMX Process Mediation Based on Choreographies. In *Proceedings of the BPM'2005 Workshops*, pages 130–143, Nancy, France, January 2006. Springer Verlag.

[10] M. Dumas, M. Spork, and K. Wang. Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In *Proceedings of the 4th International Conference on Business Process Management, Vienna, Austria*, pages 65–80, 2006.

[11] H.Foster, S.Uchitel, J.Magee, and J.Kramer. Tool Support for Model-Based Engineering of Web Service Compositions. In *IEEE International Conference on Web Services (ICWS)*, pages 95–102, Orlando FL, USA, July 2005. IEEE Computer Society.

[12] P. Inverardi, L. Mostarda, M. Tivoli, and M. Autili. Synthesis of Correct and Distributed Adaptors for Component-Based Systems: An Automatic Approach. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), Long Beach, CA, USA*, pages 405–409, 2005.

[13] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, Boston MA, USA, 2001.

[14] A. Mocan and E. Cimpian. WSMX Working Draft, Oct 2005. `http://www.wsmo.org/TR/d13/d13.3/v0.2/#L2885`.

[15] H. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada*, pages 993–1002, 2007.

[16] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *Very Large Databases*, 10(4):334–350, 2001.

[17] H. Schmidt and R. Reussner. Generating adapters for concurrent component protocol synchronisation. In *Proceedings of the 5th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 213–229, Enschede, The Netherlands, Mar 2002. Kluwer Academic Publishers.

[18] D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):292–333, 1997.