



COVER SHEET

**Dumas, Marlon and Wang, Kenneth W.S. and Spork, Murray L. (2006)
Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation.
In Dustdar, Schahram and Fiadeiro, Jose-Luis and Sheth, Amit, Eds. Proceedings
4th International Conference on Business Process Management 4102/2006,
pages pp. 65-80, Vienna, Austria.**

Copyright 2006 Springer _____

Accessed from <http://eprints.qut.edu.au>

Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation

Marlon Dumas¹, Murray Spork² and Kenneth Wang¹

¹ Queensland University of Technology, Australia
(m.dumas,kw.wang)@qut.edu.au

² SAP Research Center Palo Alto, USA
murray.spork@sap.com

Abstract. The proliferation of services on the web is leading to the formation of service ecosystems wherein services interact with one another in ways not necessarily foreseen during their development or deployment. A key challenge in this setting is service mediation: the act of retrofitting existing services by intercepting, storing, transforming, and (re-)routing messages going into and out of these services so they can interact in unforeseen manners. This paper addresses a sub-problem of service mediation, namely service interface adaptation, that arises when the interface that a service provides does not match the interface that it is expected to provide in a given interaction. The paper focuses on reconciling mismatches between behavioural interfaces, i.e. interfaces that capture ordering constraints between interactions. It presents a declarative approach to service interface adaptation based on: (i) an algebra over behavioural interfaces; and (ii) a visual language that allows pairs of provided-required interfaces to be linked through algebraic expressions. These expressions are fed into an execution engine that intercepts, buffers, transforms and forwards messages to enact the adaptation logic.

1 Introduction

There is an increasing acceptance of Service-Oriented Architectures (SOAs) as a paradigm for integrating software applications within and across organisational boundaries. In SOAs, independently developed and operated applications are made available as *services* that may be interconnected with one another using standardised protocols and languages. One of the cornerstones of SOAs is the principle that each service operates according to an interface. In a broad sense, a service's interface captures the types of messages that the service can produce and consume, the message encodings and transfer protocols that the service supports or requires, and the dependencies between message exchanges. Armed with such information, developers can build systems that draw upon functionality from multiple services and make them collaborate in complex manners.

Services may be reused across development projects, development teams, or even across organisational boundaries. It is thus normal to expect that services will be reused in context for which they were not originally designed. Consider

a procurement service which, after sending an order to an order management service, expects to receive one and only one response. Now, consider the case where this procurement service is required to engage in a new collaboration wherein the order management service may send a first response acknowledging the order and accepting or rejecting a subset of its line items, and later on send zero, one or more additional updates to accept or reject the remaining line items as their availability is determined. This interface mismatch is illustrated in Figure 1. The figure shows an interface provided by an existing service (the *provided interface*) and the interface that this service is expected to provide in a new context (the *required interface*). The interfaces shown in this example are taken from industry standards: the provided interface corresponds to a fragment of an xCBL/UBL *order management process*³ while the required interface corresponds to a RosettaNet *partner interface process*.⁴

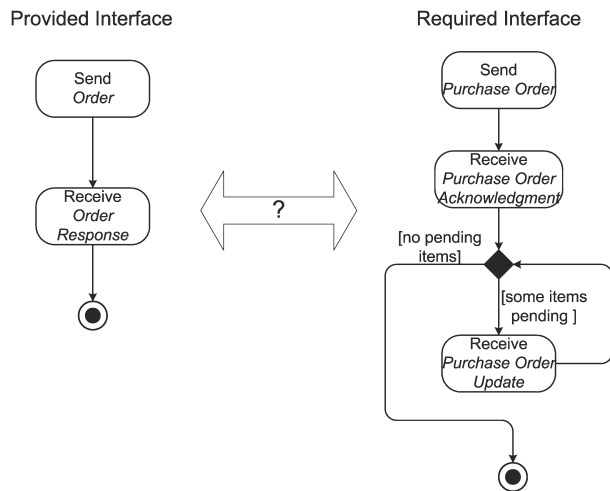


Fig. 1. Interface mismatch scenario

Cast more generally, service reuse leads to situations where a service is required to participate in multiple collaborations where different interfaces are required from it. These “required interfaces” may correspond to different message granularities, message types, and dependencies between message exchanges. Thus, service reuse calls for mechanisms to mediate between the interface natively provided by a service and the various interfaces that are required from it. We call this problem *service interface adaptation*.

Service interfaces can be described from a structural perspective, where the focus is on message types, and from a behavioural perspective, where the focus is on control dependencies between message exchanges. The problem of interface adaptation from the structural perspective has received considerable attention, leading to a number of transformation definition (e.g. XSLT) and schema

³ <http://www.xcbl.org> and <http://docs.oasis-open.org/ubl/prd-UBL-2.0>

⁴ <http://www.rosettanet.org>

mapping tools such as Microsoft BizTalk Mapper, Stylus Studio XML Mapping Tools, and SAP XI Mapping Editor.⁵ In comparison, the problem of interface adaptation from a behavioural perspective is still open.

In this setting, the research question that we address can be formulated as follows: how to enable a service implementing a given behaviour (e.g. the behaviour on the left-hand side of Figure 1) to participate in interactions where a different behaviour, yet the same functionality, is required from it (e.g. the behaviour on the right-hand side of Figure 1). Traditionally, this problem is addressed by developing adaptors using programming languages. However, these adaptors are costly to develop and to maintain. Furthermore, the use of programming languages makes it difficult to check that these adaptors correctly implement the intended adaptation logic or that they do not create deadlocks.

Accordingly, we propose a declarative approach to service interface adaptation that emphasises on the behavioural perspective and can coexist with existing approaches to structural interface adaptation. The proposal comprises a visual notation underpinned by an algebra of interface transformation operators. The visual notation provides a declarative means for developers to map between required and provided interfaces. The algebra provides a semantics for the notation and provides a basis for executing these mappings. The proposal has been validated by a prototype tool that mediates between pairs of provided-required interfaces by intercepting, buffering, transforming and forwarding messages according to interface transformation expressions.

The rest of the paper is structured as follows. Section 2 introduces background concepts. Next, Section 3 presents the algebra of interface transformation operators while Section 4 presents the visual notation and its relationship to the algebra. Section 5 then discusses a prototype implementation. Finally, Section 6 compares our proposal with related work and Section 7 concludes.

2 Background

The operators put forward in this paper are defined over *behavioural interfaces*. We view a behavioural interface as a collection of control dependencies defined over a set of message exchanges. Behavioural interfaces complement structural interfaces such as those that can be described in WSDL. Structural interfaces describe the individual message exchanges in which a service can engage (e.g. in terms of message types and transport protocols) while behavioural interfaces are concerned with dependencies between message exchanges. Behavioural interfaces are known under different names, including *abstract process* in BPEL [7] and *collaboration protocol profile/agreement* in ebXML [10].

Various languages can be used to specify behavioural interfaces, e.g. UML Activity Diagrams, BPMN [12] or BPEL. We abstract from the language employed to describe behavioural interfaces by adopting a general definition based notions from the field of concurrency theory. For illustration purposes however,

⁵ See <http://www.biztalk.org>, <http://www.stylusstudio.com>, and <http://www.sap.com/platform/netweaver/components/xi> resp.

we depict behavioural interfaces using UML activity diagrams in which actions are named according to the type of message being sent or received.

Behavioural interfaces are defined in terms of *communication action schemas*. A communication action schema⁶ is a statement that a service may send or receive a message of a given type. We represent a communication action as a tuple (AN, D, MT) where AN is the name of the action, D indicates whether the action is inbound (receive) or outbound (send) with respect to the service being described, and MT denotes the type of messages that are sent or received by the action. Since the focus is on behavioural aspects, we abstract from the way message types are represented and instead we refer to message types through identifiers. For example, a communication action whereby a procurement service sends a purchase order to an order management service is represented as a tuple (“place order”, “purchase order”, out).

Formally, we define a behavioural interface as a possibly infinite set of traces (or strings) over an alphabet made up of communication actions. A trace t over an alphabet of communication actions defines a linear order and we call this order relation $<_t$. Each token in a trace represents an *instance of a communication action*. Thus, we distinguish between a communication action schema as defined above and instances (i.e. occurrences) thereof that appear in a trace. A trace may contain several instances of the same communication action schema. This is the case of behavioural interfaces that define repetitive behaviour (e.g. the interface on the right-hand side of Figure 1) such that the same action may be executed more than once as part of a single execution of the behavioural interface (e.g. action “Receive PO Update” in Figure 1). Below, we represent traces as lists of communication action instances $[a_1, \dots, a_n]$.

Different traces of an interface may include instances of different actions. This happens when there is conditional branch in the interface. For example, it may be that for purchase orders with quotes > 500 something is done, while for purchase orders with lower quotes, something else is done. However, we can group the traces of an interface into disjoint groups gt_1, gt_2, \dots such that all the traces in a given group gt_i contain the same set of action instances, albeit ordered differently in each trace. Given a group of traces gt of an interface I , we define a run r over interface I as a partial order $<_r$ such that:

$$\forall a_1, a_2 \in Actions(gt) \ a_1 <_r a_2 \leftrightarrow (\forall t \in gt \ a_1 <_t a_2)$$

...where $Actions(gt)$ denotes the set of action instances common to all traces in group gt . If $a_1 <_r a_2$ we say that a_1 *necessarily precedes* a_2 .

Consider for example the interface represented in Figure 2. It consists of four traces: $t_1 = [a_1, a_2, a_4]$, $t_2 = [a_1, a_3, a_5, a_6, a_7]$, $t_3 = [a_1, a_3, a_6, a_5, a_7]$ and $t_4 = [a_1, a_3, a_6, a_7, a_5]$.⁷ We can cluster these traces into two groups $gt_1 = \{t_1\}$ and $gt_2 = \{t_2, t_3, t_4\}$ such that each group corresponds to a run. The run r_1

⁶ We write *communication action* or simply *action* where there is no ambiguity.

⁷ Throughout the paper, we use lowercase to denote action instances and uppercase to denote action schemas. For example, a_1 denotes an instance of action A_1 .

corresponding to gt_1 is such that $a_1 <_{r_1} a_2$ and $a_2 <_{r_1} a_4$, while the run r_2 corresponding to gt_2 is such that $a_1 <_{r_2} a_3$, $a_3 <_{r_2} a_5$, $a_3 <_{r_2} a_6$, and $a_6 <_{r_2} a_7$.

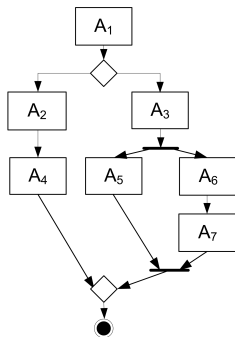


Fig. 2. Sample behavioural interface used to illustrate the notions of trace and run

3 Interface Transformation Algebra

The proposed model for interface transformation is based on a collection of operators for expressing how to go from one behavioural interface to another. We propose six operators namely *flow*, *scatter*, *gather*, *collapse*, *burst*, and *hide*. We do not claim that this set of operators is complete in any sense. However, we have designed each operator based on common mismatch pattern identified in prior work. Specifically, the flow, scatter gather, and hide operators correspond to the mismatch patterns identified in [9, 1, 3]⁸, the collapse operator corresponds to the “bundling patterns” supported in SAP XI (see Section 6) while the burst performs the opposite of the collapse.

All six operators are algebraic in the sense that they take as input a behavioural interface (and other parameters) to produce another behavioural interface. In the sequel, an interface taken as input by a transformation operation is called the *source interface* while the interface that is produced is called the *target interface*. The notion of source and target interface are not to be confused with those of provided and required interfaces. The source interface may correspond to the required interface, the provided interface, or to an intermediate interface generated by another operation as illustrated later.

To define the transformation operators, we use the following notations:

- *Interface* denotes the type of all possible behavioural interfaces.
- *Action* $\langle T \rangle$ denotes the type of all possible actions that produce or consume a message of type T . This is a parameterised type.
- *AID* denotes the type of all *action identifiers*.
- *direction*(a) denotes the directionality of action a (inbound or outbound).

⁸ The hide operator also corresponds to notions of behaviour abstraction studied in the area of behaviour inheritance [11].

3.1 The Flow Operator

The Flow operator describes a transformation where an action defined in the source interface becomes another action in the target interface. The type of this operator is:

$$Flow : Interface, Action \langle ST \rangle, (ST \rightarrow TT), AID \rightarrow Interface$$

The Flow operator takes as input: (i) a source interface SI , (ii) an action SA within this source interface that produces or consumes a message of a type ST , (iii) a function F that converts a message of type ST to a message of another type TT , and (iv) an action identifier ID_{TA} . From there, it produces an interface TI which has the same set of runs as SI except that in each of the resulting runs, every instance of action SA is replaced by an instance of an action TA , such that $direction(SA) = direction(TA)$. The message produced by an instance of TA that replaces an instance of SA (say sa) is obtained by applying function F to the message produced or consumed by sa . Thus, $TA = (ID_{TA}, TT, direction(SA))$. The use of the Flow operator is illustrated in Figure 3.

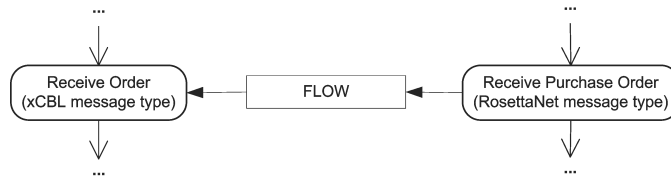


Fig. 3. The Flow operator

3.2 The Gather Operator

The Gather operator is applied when multiple actions from the source interface map to a single action in the target interface. The messages produced by the designated actions in the source interface are combined together using an aggregation function. The use of this operator is illustrated in Figure 4.

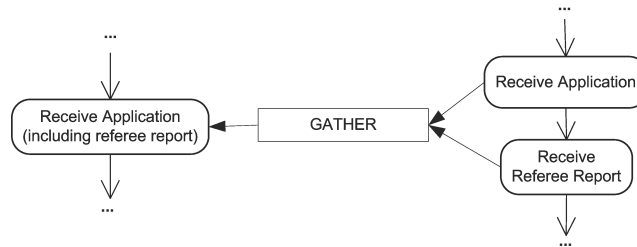


Fig. 4. The Gather operator

The Gather operator is in fact an infinite family of operators $(Gather)_n$ ($n \geq 2$) with the following type:

$$\begin{aligned} Gather_n : & \text{Interface}, \text{Action} \langle ST_1 \rangle \dots \text{Action} \langle ST_n \rangle, \\ & (ST_1 \dots ST_n \rightarrow TT), AID \rightarrow \text{Interface} \end{aligned}$$

$Gather_n$ takes as input: (i) an interface SI , (ii) n actions SA_1, \dots, SA_n , (iii) an aggregation function AF , (iv) an action identifier ID_{TA} . The resulting interface TI defines the same set of runs as SI except that in each of these runs, every *consecutive combination* of instances of actions $SA_1 \dots SA_n$ (say $sa_1 \dots sa_n$) is replaced by an instance of an action TA (say ta such that ($TA = ID_{TA}, TT, direction(SA)$)). Instance ta is placed in the resulting run such that:

$$\begin{aligned} \forall a \in \text{Actions}(r) \setminus \{sa_1, \dots, sa_n\} (\exists i \in [1..n] a <_r sa_i) \Rightarrow a <_{r'} ta \\ \wedge (\forall i \in [1..n] a >_r sa_i) \Rightarrow a >_{r'} ta \end{aligned}$$

... where r is the original run and r' is the run obtained after replacement of $sa_1 \dots sa_n$ with ta . Runs r and r' are identical except for this replacement.

The message produced by an action instance ta that replaces a combination of action instances $sa_1 \dots sa_n$ is obtained by applying aggregation function AF to the list of messages produced or consumed by $sa_1 \dots sa_n$.

By *consecutive combination* of instances of actions $SA_1 \dots SA_n$ in a run r , we mean that in between an occurrence of SA_i and an occurrence of SA_{i+1} (where $i \in [1..n-1]$), there is no occurrence of another action SA_j ($j \in [1..n]$) such that $sa_i <_r sa_j <_r sa_{i+1}$. For this definition to make sense, the following precondition must be associated to operator $Gather_n$: the set of actions SA_1, \dots, SA_n should be ordered in a way compatible with their control dependencies in the source interface SI . Specifically, for any given consecutive combination of actions as defined above, the following must hold:

$$\forall i, k \in [1..n], i < k \rightarrow \forall r \in \text{Runs}(SI) \neg (sa_k <_r sa_i)$$

Another precondition of the $Gather_n$ operator is that all the actions being gathered should have the same directionality, and there should not be an action of the opposite directionality that lies in-between the actions being gathered. Formally:

$$\begin{aligned} \forall i, j \in [1..n], i \neq j \rightarrow (\forall r \in \text{Runs}(SI) sa_i \in \text{Actions}(r) \rightarrow sa_j \in \text{Actions}(r)) \\ \forall i, j \in [1..n], i < j \rightarrow \text{Direction}(sa_i) = \text{Direction}(sa_j) \wedge \\ \neg \exists sa_m \in \text{Actions}(SI) \text{Direction}(sa_m) \neq \text{Direction}(sa_i) \wedge \\ sa_i <_r sa_m <_r sa_j \end{aligned}$$

The rationale for this precondition is the following. $Gather_n$ replaces a combination of actions $sa_1 \dots sa_n$ with a single action ta . If $sa_1 \dots sa_n$ were “receives” and there was a “send” (say $sendA$) between them ($sa_1 <_r sendA <_r sa_n$), we would have that $sendA <_{r'} ta$ due to $sendA <_r sa_n$ and the definition of $Gather_n$. However, the service implementing SI can not execute $sendA$ prior to the execution of ta , since the execution of $sendA$ requires information coming from sa_1 ($sa_1 <_r sendA$) and this information is only known once ta has been executed. So on the one hand $sendA$ needs to occur before ta and on the other hand it needs to occur after ta . The above precondition prevents this deadlock.

3.3 The Scatter operator

The Scatter operator is applied when a single action in the source interface is transformed into multiple actions in the target interface.

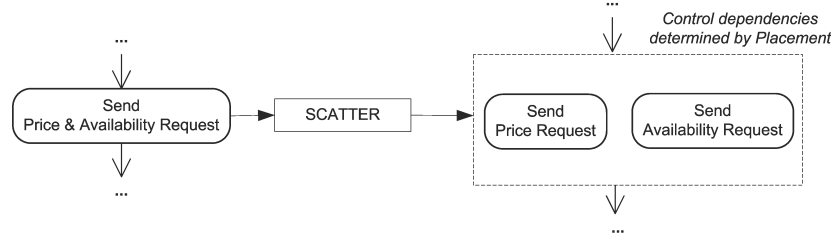


Fig. 5. The Scatter operator

Like with the Gather, $(Scatter)_n$ ($n \geq 2$) is an infinite family of operators: $Scatter_2, Scatter_3, \dots$. For a given n , the type of this operator is:

$$Scatter_n : Interface, Action\langle ST \rangle, (ST \rightarrow TT_1 \dots TT_n), \\ Placement\langle Action\langle TT_1 \rangle \dots Action\langle TT_n \rangle \rangle \rightarrow Interface$$

Operator $Scatter_n$ takes as parameter an interface SI , an action SA , a function DS that splits a message into multiple ones, and a partially ordered set of actions $TA_1 \dots TA_n$ (called a *placement*) all with the same directionality, and returns an interface. The resulting interface TI has the same set of runs as SI except that in every run of SI , every instance of SA is replaced by a subrun containing instances of actions $TA_1 \dots TA_n$. The actions in the subrun are arranged as described by the placement P . The placement may be represented in many ways. One possible representation (though not necessarily the most expressive one) is as an expression composed using operators SEQ and PAR that represent *sequential* and *parallel* placement respectively. For example given a placement $SEQ(TA_2, PAR(TA_1, TA_3))$, each occurrence of SA is replaced by a subrun in which TA_2 is executed first followed by both TA_1 and TA_3 in any order.

The messages produced or consumed by the instances of actions $TA_1 \dots TA_n$ that replace an instance of SA (say sa), are obtained by applying the data splitting function DS to the message produced or consumed by sa .

3.4 The Collapse Operator

The Collapse operator is used when a stream of messages resulting from multiple instances of the same communication action is aggregated into a single message, as illustrated in Figure 6. In this figure, the source interface (left) is such that the shipment notifications are sent incrementally as the products are dispatched. Meanwhile, the target interface (right) requires a single shipment notification.

The type of the Collapse operator is:

$$Collapse : Interface, Action\langle ST \rangle, (List\langle ST \rangle \rightarrow TT), AID \rightarrow Interface$$

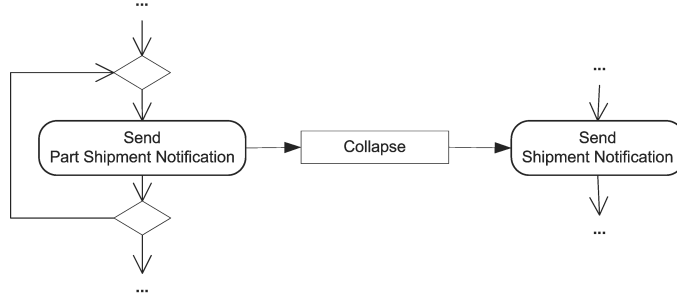


Fig. 6. The Collapse operator

The Collapse operator takes as parameter an interface SI , an action SA , an aggregation function AF , and an action identifier ID_{TA} , and produces a target interface TI . The resulting interface TI has the same set of runs as SI except that in each run, the set of instances of SA (if any) is replaced by a single instance of action TA such that $TA = (ID_{TA}, TT, direction(SA))$. The message produced or consumed by an instance of TA that replaces a sequence of instances of SA (say $sa_1 \dots sa_n$) is obtained by applying the aggregation function AF to the set of messages produced or consumed by $sa_1 \dots sa_n$.

The collapse operator requires the execution environment to: (i) track the progress of the source and target interfaces; (ii) perform a reachability analysis each time the source interface changes state;⁹ (iii) once the action to be collapsed is no longer reachable from the current state, apply the aggregation function to the set of accumulated messages; (iv) dispatch the aggregated message when the target interface reaches a state where it can consume it.

The collapse operator as defined above is such that all instances of a “source” action are replaced by a single action instance. In some scenarios however, one may wish not to aggregate all instances of the source action, but only a subset thereof up to the point where a milestone is reached. For example, one may need to aggregate all part shipment notifications until an invoice is received, then aggregate the next set of shipment notifications until another invoice is received and so on. In future, we plan to investigate extensions to the Collapse operator that capture more general scenarios.

3.5 The Burst Operator

The Burst operator works in the reverse of the Collapse operator and is used when a single message needs to be split into a stream of messages. This operator is used where the transformed stream of message consists of repeated instances of the same communication action as illustrated in Figure 7.

The type of the Burst operator is:

$$Burst : Interface, Action \langle ST \rangle, (ST \rightarrow List \langle TT \rangle), AID \rightarrow Interface$$

⁹ We can optimise this step so that the analysis is only performed once per state.

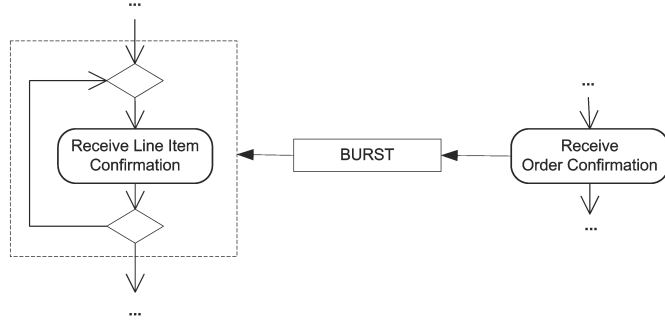


Fig. 7. The Burst operator

The operator Burst takes as parameter an interface SI , an action SA from SI , a function SF , and an action identifier ID_{TA} , to produce a target interface TI . The resulting interface TI has the same set of runs as SI except that in each run, every instance of action SA is replaced by a sequence of instances of an action TA such that $TA = (ID_{TA}, TT, direction(SA))$. The message produced by a sequence of instances of TA ($ta_1 <_r ta_2 <_r \dots ta_n$) that replaces a single instance of action SA (say sa) is obtained by applying the “splitting” function SF given as third parameter of the Burst operator, to the message produced or consumed by sa .

3.6 The Hide Operator

The Hide operator is used when an action from the source interface is not required in the target interface. Specifically, the action produced by the source interface is to be ignored (i.e. discarded) as illustrated in Figure 8.

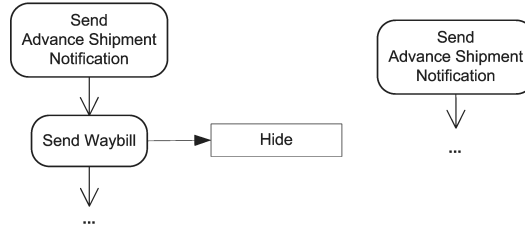


Fig. 8. The Hide operator

The type of the Hide operator is:

$$Hide : Interface, Action\langle ST \rangle \rightarrow Interface$$

The Hide operator takes as input an interface SI and an action SA within this interface, and produces as output an interface TI identical to SI , except that in each run of SI , any instance of action SA is removed. Before applying this

operator, the developer needs to ensure that the message produced or consumed by the action being hidden is not crucial to the operation of the adapted service.

We have intentionally avoided introducing any operators that handle the scenario where an action from the target interface is needed but is not provided by the source interface. This scenario requires the introduction of business logic in the adaptor, which is undesirable from a software maintenance perspective. Indeed, this would result in the business logic being spread across the service and the adaptors. Subsequently, any change in the business logic would require developers to trace back which adaptors need to be changed.

4 Visual Notation

4.1 Visual Representation of Mapping Expressions

An interface mapping between a provided interface and a required interface is a collection of interface transformation expressions (E_1, \dots, E_n). A transformation expression can be either outbound (dealing with “send” actions) or inbound (dealing with “receive” actions).

An interface transformation expression is represented as follows. Each operation in the expression a node linked through edges to other operations or to actions in the required interface or in the provided interface. Edges are directed according to the message flow. Visually, we distinguish two groups of operators: Hide, Flow, Gather, Scatter on the one hand, and Burst and Collapse on the other. Nodes corresponding to the first group can be represented by the same symbol (say a rectangle). They can be distinguished because a Flow node has one incoming and one outgoing edge, a Gather node has multiple incoming and one outgoing edge, a Scatter node has one incoming and multiple outgoing edges, and a Hide node has multiple incoming edges and no outgoing ones. The Collapse and Burst nodes have one incoming and one outgoing edge, so to distinguish them from the Flow, we need to use different symbols. We represent them as concentric rectangles containing two convergent or divergent arrows indicating whether it is a collapse or a burst respectively.

Figure 9 illustrates how interface transformation expressions are visually represented using the example introduced in Section 1. The mapping expressions (namely E_1 and E_2) captured in this figure can be textually expressed as follows:

$$E_1 = Flow(PI, PA_1, F_1, RA_1)$$

$$E_2 = Gather(Collapse(RI, RA_3, F_2, IA), RA_2, IA, F_3, PA_2)$$

The outbound interface mapping expression E_1 is a single-operator transformation expression that converts action PA_1 into RA_1 . The inbound interface mapping expression E_2 is a composition of a Gather and a Collapse operator. The Collapse operator is applied first and transforms RA_2 into an intermediate interface containing an action IA that replaces action RA_3 . The interface obtained from the Collapse operation is then given as input to the Gather function which merges RA_2 and IA and replaces them with action PA_2 .

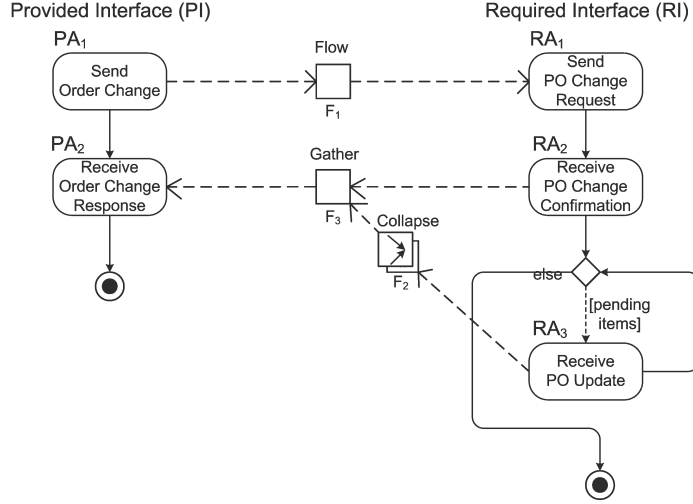


Fig. 9. Example of a visual mapping

Formally, an interface transformation expression is a directed acyclic graph whose sources are actions in one interface (e.g. the required) and whose sinks are actions in the other interface (e.g. the provided interface). For an interface mapping to cover all possibilities, it should be such that every action in the provided interface is the source or the sink of at least one transformation expression.

4.2 Mapping Constraints

Interface mappings may create deadlocks. To detect such deadlocks, we define below a condition that an interface mapping needs to fulfil. We do not claim that this condition covers all possible deadlock scenarios. In future work, we plan to investigate more general conditions.

Given any two expressions $E_i, E_o \in IM$ such that E_i is inbound and E_o is outbound, if there are four actions $A_1 \in TargetActions(E_i)$, $A_2 \in SourceActions(E_o)$, $A'_1 \in SourceActions(E_i)$ and $A'_2 \in TargetActions(E_i)$, then the precedence relation between A_1 and A_2 , if any, should be compatible with that between A'_1 and A'_2 . Specifically, for every run r of the target interface of E_i such that r contains an instance of action A_1 (say a_1), let a'_1 be an instance of action A'_1 that maps to a_1 through expression E_i . Now assuming that in r there is an instance of A_2 (say a_2) that maps to an instance of A'_2 (say a'_2) through expression E_o , then:

$$a_1 < a_2 \Rightarrow \neg(a'_2 < a'_1)$$

Figure 10 shows a violation of this constraint. The provided interface expects to receive the payment (A_1) before sending the shipment A_2 , while in the required interface the opposite holds, thus creating a deadlock. More generally, the rationale for this constraint is that it is not possible to send information that is dependent on other information we have not yet received, nor to receive information that is dependent on other information we have not yet sent.

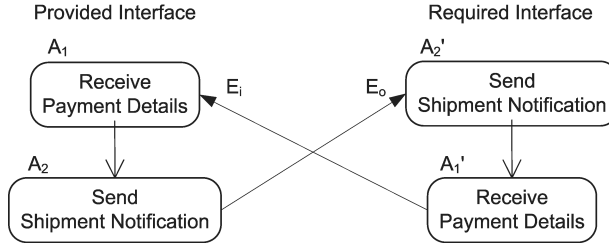


Fig. 10. Example violating the first mapping constraint

5 Tool Support

We are currently developing a prototype implementation of an *interface mapping tool* and a *service mediation engine* that support the visual notation and the algebra respectively. The implementation of the mediation engine has been completed while that of the mapping tool is underway.

The mapping tool is a graphical editor allowing developers to load pairs of provided-required behavioural interfaces and to link them through interface transformation expressions. Behavioural interfaces are represented as BPEL abstract processes supplemented by their corresponding WSDL definitions. Data manipulation functions are coded in XSLT. This provides a hook for connecting the editor with schema mapping tools that produce XSLT as output.

The output of the interface mapping tool consists of the original pair of required-provided interfaces, the transformation expressions specified by the developer, as well as configuration information related the service endpoints that implement the provided and the required interfaces. In line with our aim to abstract away from the language used to describe behavioural interfaces, the engine relies on an abstract representation of behavioural interfaces in the form of Finite State Machines (FSMs) whose transitions are labelled by communication actions. Such FSMs capture the information needed to execute the transformation expressions while abstracting away from evolving technology such as BPEL. This design choice entails however that, when deploying an interface mapping, the mapping tool must convert the BPEL abstract processes that it takes as input into the FSMs used by the execution engine. Translations from BPEL process definitions to FSMs have been studied in the literature, see e.g. [6].

The deployment of an interface mapping into the mediation engine results in the engine exposing a new service endpoint that behaves according to the required interface. An external application or service (say S_1) can then send messages to this endpoint managed by the mediation engine which, based on the logic of the corresponding transformation expressions, stores, transforms and eventually forwards messages to the service endpoint that implements the required interface (say S_2). Subsequently, the engine intercepts all messages between S_1 and S_2 and manipulates them according to the transformation expressions.

Messages intercepted by the engine need to be correctly associated to their corresponding service instance. To this end, we impose that every SOAP message intercepted by the mediation engine should contain a WS-Addressing *messageID*

and (optionally) a *relatesTo* header. The engine uses these headers to correlate new messages with previously intercepted messages in order to determine the correct service instance to which the new message belongs. Messages with a *relatesTo* header are assigned to an existing service instance, while messages without this header lead to the creation of new instances, unless there is no service registered with the mediation engine that matches the action identifier of the message (SOAP-Action header), in which case the message is put into a pool of unallocated messages. The mediation engine includes an administration console to monitor the current status of service instances managed by the engine and to view histories of intercepted, transformed and forwarded messages.

6 Related Work

Traditionally, the concept of “interface” has been associated to a collection of operations or message type definitions. This view has transpired into WSDL. Accordingly, the problem of interface adaptation has been approached as a schema reconciliation problem. In the case of Web services, this comes down to mapping between different XML schemas which is a well-understood problem [8].

In this paper, we adopt a broader view on interfaces, encompassing behaviour in addition to structure. This view has been advocated in the field of component-based software engineering where the issue of interface adaptation over behavioural interfaces has received some attention. Yellin & Strom [13] define a notion of compatibility of components whose behavioural interfaces (called *protocols*) are described as FSMs. Their work addresses the question of verifying that a given adaptor (specified as a FSM) is able to reconcile two incompatible behavioural interfaces. The authors assume that the adaptors can not store an unbounded number of messages. Our Collapse operator breaks this assumption. For example, the adaptor specified in Figure 9 needs to store an unbounded number of “updates”. The “bounded buffer” assumption is motivated by undecidability issues that arise when verifying properties of adaptors. But as shown in Figure 9, the assumption is unrealistic in the application domain of Web services. Yellin & Strom also discuss how to generate an adaptor from a set of links between *parameters* (i.e. message parts) in the provided interface and corresponding parameters in the required interface. But there is an assumption that the adaptors do not use the equivalent of a Collapse, Burst, or Hide operator.

Another technique for generation of adaptors for behavioural interfaces is defined in [9]. As in Yellin & Strom, the authors deal with mismatches corresponding to the “Flow”, “Gather” and “Scatter” operators, not with “Burst”, “Collapse” and “Hide”. This work also differs from ours in that it does not consider the use of composable transformation operators with a graphical syntax.

More recent research has addressed the problem of interface adaptation in the context of Web services. Benatallah et al. [3] identify a set of “mismatch patterns” between behavioural interfaces and provide templates of BPEL code that developers may reuse to build adaptors that resolve these mismatches. However, the compositionality of these BPEL templates is not considered and thus the ap-

proach is not systematic. Similar mismatch patterns are identified in [4] and [1] where high-level architectures for addressing such mismatches are proposed. The ADAPT framework [1] goes further by proposing a notation for N-to-M mappings, i.e. mappings where data coming from N services are collected and repartitioned among M services. This is similar to the Gather and Scatter operators but it does not take into account any information contained in the behavioural interfaces, e.g. the data is forwarded to the target services as soon as it has been collected and in no particular order, whereas our Gather operator forwards messages in a specific order to fulfill the constraints of the target interface. Altenhofen et al. [2] propose a formal model for process mediation based on Abstract State Machine (ASM) specifications. They show how these ASMs can be refined to deal with mismatch patterns such as those identified in [4]. Fuchs [5] proposes another approach to interface adaptation. However, this contribution focuses on reconciling operational differences such as security policies, service level agreement, etc.

SAP eXchange Infrastructure (XI) supports behavioural interface adaptation through so-called “bundling patterns”¹⁰. These patterns come with process templates that can be used in scenarios where certain types of messages need to be buffered until they are all available and then aggregated into a single message. However, these patterns only address a restricted set of behavioural interface adaptation scenarios and do not provide a systematic approach to the problem.

7 Conclusion and future work

In this paper, we introduced a declarative approach to service interface adaptation based on an algebra of six operators over behavioural interfaces; and a visual language that allows pairs of provided and required interfaces to be linked through algebraic expressions. The paper also introduced an architectural view of our execution engine that consumes these algebraic expressions and facilitates message interception, buffering and transformation to enact the adaptation logic.

In future work, we plan to investigate notions of completeness in the context of service interface adaptation that would enable us to characterise the expressiveness of the algebra and to define more powerful extensions or alternatives thereof. One fundamental question that should be addressed is: When can a service implementing a given provided interface be adapted to fit a given required interface without adding new business logic into the adaptor. As discussed in Section 3.6, adding business logic into the adaptors can lead to maintainability issues, since business logic would then be spread across the service and its adaptors, rather than being concentrated in the service. In other words, we envisage that adaptors should be restricted to data transformations and coordination aspects, leaving the business logic entirely within the service.

In addition, we plan to develop techniques to semi-automatically infer possible links between provided and required interfaces. For example, when a send action in a provided interface has an associated message type similar (according to a similarity metrics) to that of a send action in the required interface, we

¹⁰ See <http://tinyurl.com/h427a> and <http://tinyurl.com/kpe3a>.

can infer that these two actions should be linked through a Flow operation. By combining these heuristics with similar heuristics developed in the context of schema mapping [8], we seek to design techniques for semi-automatic generation of adaptors for conversational services.

Acknowledgment This research is funded by a Queensland “Smart State” Fellowship and ARC Linkage Project LP0455394, both co-sponsored by SAP.

References

1. G. Alonso, C. Pautasso, and B. Björnstad. CS Adaptability Container. Deliverable #11, EU FP5 Project “ADAPT”, August 2004.
2. M. Altenhofen, E. Börger, and J. Lemcke. An abstract model for process mediation. In *In Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM)*, pages 81–95, Manchester, UK, November 2005. Springer.
3. B. Benatallah, F. Casati, D. Grigori, H. R. 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. Springer, 2005.
4. E. Cimpian and A. Mocan. WSMX Process Mediation Based on Choreographies. In *Proceedings of the Business Process Management Workshops*, pages 130–143, Nancy, France, September 2005. Springer.
5. M. Fuchs. Adapting web services in a heterogeneous environment. In *Proceedings of the Second IEEE International Conference on Web Services, ICWS 2004, San Diego, California, USA*, pages 656–664, 2004.
6. 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)*, Orlando FL, USA, July 2005. IEEE Computer Society.
7. R. Khalaf, N. Mukhi, F. Curbera, and S. Weerawarana. The Business Process Execution Language for Web Services. In *Process-Aware Information Systems*. John Wiley & Sons, 2005.
8. L. Popa, Y. Velegrakis, R. Miller, M. Hernández, and R. Fagin. Translating web data. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, pages 598–609, Hong Kong, China, August 2002.
9. 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)*, Enschede, The Netherlands, March 2002. Kluwer Academic Publishers.
10. UN/CEFACT and OASIS. ebXML Business Process Specification Schema (Version 1.01). <http://www.ebxml.org/specs/ebBPSS.pdf>, 2001.
11. W. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
12. S. White. Business Process Modeling Notation (BPMN). Version 1.0 - May 3, 2004, BPMI.org, 2004. www.bpmi.org.
13. D. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.