Enforcing web services business protocols at run-time: a process-driven approach

Biörn Biörnstad*, Cesare Pautasso and Gustavo Alonso

Department of Computer Science
ETH Zürich
CH-8092 Zürich, Switzerland
E-mail: bioernstad@inf.ethz.ch
E-mail: pautasso@inf.ethz.ch
E-mail: alonso@inf.ethz.ch
*Corresponding author

Abstract: Business processes provide abstractions for modelling business protocols that define the correct interactions between two or more Web services (WS). It has been shown that it is possible to automatically derive role-specific processes from a global protocol definition and also statically verify the compliance of a local process with the corresponding global business process. In this paper, we show that a similar approach can be used at run-time. We propose to employ process-based tools to enforce that the messages exchanged between different WS comply with a given business protocol, both in terms of sequencing constraints and data flow characteristics. Our solution simplifies the implementation of WS because it helps to separate the concern of business protocol compliance from the actual service implementation. To do so, we show how to transparently add a protocol enforcement layer to the WS messaging stack. Our experimental results indicate that this imposes a minimal overhead.

Keywords: Web services (WS); choreography; run-time protocol enforcement; business protocols; conversation controller.

Reference to this paper should be made as follows: Biörnstad, B., Pautasso, C. and Alonso, G. (2006) 'Enforcing web services business protocols at run-time: a process-driven approach', *Int. J. Web Engineering and Technology*, Vol. 2, No. 4, pp.396–411.

Biographical notes: Biörn Biörnstad is a PhD student in the Department of Computer Science at ETH Zurich. He received a degree in Computer Science from ETH Zurich in 2002. His research interests are in the area of workflow management and the intersection of message-based and streaming communication.

Cesare Pautasso is a Senior Researcher in the Department of Computer Science at ETH Zurich. He completed his graduate studies with a PhD from ETH Zurich in 2004 and his undergraduate studies at Politecnico di Milano, Italy with a Computer Science Engineering Degree (cum laude) in 2000. His research interests focus at the intersection of autonomic computing, business process management, cluster/grid computing and visual languages, with the goal of exploring innovative techniques for building large-scale distributed systems by means of software composition. His research ideas have been driving the development of the JOpera for Eclipse system (www.jopera.org). In

addition to doing research, Pautasso has been active giving lectures (both in industry and at the university) on the topics of Middleware, Web Services and Service-Oriented Architectures.

Gustavo Alonso is Professor in the Department of Computer Science at ETH Zurich. He holds degrees in Telecommunications Engineering from the Madrid Technical University (1989) and in Computer Science (MS 1992, PhD 1994) from the University of California at Santa Barbara. After graduating, he was a visiting scientist in the IBM Almaden Research Laboratory in San Jose, California. His research interests include Web services, grid and cluster computing, sensor networks, databases, workflow management, scientific applications of database and workflow technology, pervasive computing and dynamic aspect oriented programming.

1 Introduction

Web services (WS) standards and tools provide the basic infrastructure for supporting service-oriented architectures (Weerawarana *et al.*, 2005) in enterprise application integration and electronic commerce. The idea is for WS to facilitate access to applications by providing a standard interface. These WS are then combined into more complex services. Combining simpler WS into more complex ones is often referred to as composition. In this context, workflow processes have been successfully applied to define WS compositions (Leymann *et al.*, 2002; Curbera *et al.*, 2003). Processes provide high-level abstractions for modelling a *conversation*, *i.e.*, a sequence of message exchanges between a set of WS. Likewise, workflow processes have also been used to model *business protocols* (Chiu *et al.*, 2004; W3C, 2002), *i.e.*, a set of rules which defines a correct conversation with a WS (Alonso *et al.*, 2004). Given the relationship between a composition and a business protocol, many useful results concerning the static validation of one in terms of the other have been presented (van der Aalst and Weske, 2001; Bussler, 2002; Schulz and Orlowska, 2004).

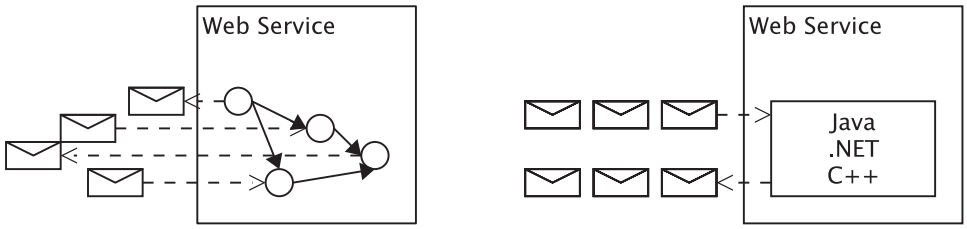
When several services interact, all of them must comply with a common protocol. In general, and especially in an open WS composition scenario, a service cannot always assume the correct behaviour of its communication partners. Thus, to protect itself, the service must enforce the protocol before taking actions based on any received message. In this paper, we use a process-based approach to enforce business protocols at run-time. This entails the verification of every message received or sent by a WS participating in a conversation. Messages are checked with respect to ordering and data format constraints. Process-based modelling techniques provide a formal description of such constraints that can be used to enforce the correctness of the interaction at run-time.

For the purposes of this paper, we assume that there exists a global model of the business protocol that defines the rules followed by all of the partners involved in a conversation. We then use a process-based system to enforce the business protocol. Every *partner* involved in the conversation is associated with the *public process* describing its view of the protocol (van der Aalst and Weske, 2001). This process defines the correct messages to be received (or sent) at every stage of the conversation. For each ongoing conversation, every partner maintains an instance of its public process. When a message is exchanged between two partners, both update the state of their process instance to

reflect the progress of the conversation. This makes it possible to check every message against the current state of the process instance in order to accept or reject it without requiring a centralised coordinator.

Given the existing work on verifying that a public process complies with a global protocol (Benatallah *et al.*, 2004) as well as on extracting role-specific views from a global business protocol definition (Schulz and Orlowska, 2004), we assume that the public process assigned to each WS correctly defines its behaviour in the conversation. However, we do not assume that each of the WS is implemented with process-based tools (Figure 1, left). If this had been the case, it would have been possible to apply some form of static verification to ensure that the private process driving the implementation of the WS complies with the aforementioned public process (Kindler *et al.*, 2000). In practice, most WS are still implemented using traditional programming languages (Figure 1, right). Thus, the implementation includes both the private business logic and the logic checking that clients of the WS comply with the public process.

Figure 1 WS with a process-based implementation (left) and an implementation in a traditional programming language (right)



In view of this, the paper makes the following contributions. First, we propose to implement the run-time protocol enforcement in a component, which is logically separate from the actual service implementation. This prevents invalid messages from triggering possibly expensive operations. Second, since the protocol enforcement is done transparently with respect to the service implementation, the conversation controller can be used without modifying existing services. Or, if an existing service implementation is changed, the conversation controller can be used to ensure the correctness of the new implementation. By separating the protocol verification from the business logic, the developer of the service can concentrate on the business logic rather than having to deal with a potentially large set of exceptional cases (Kuno and Lemon, 2001). Third, we show how to enforce protocols using a *process-based* conversation controller. In doing so, an existing process-based specification of the protocol can be reused and an error-prone manual translation to conventional code can be avoided. It is also possible to reuse an existing process management infrastructure to keep the state of conversations. Finally, we also show that our solution is efficient and does not introduce a large overhead.

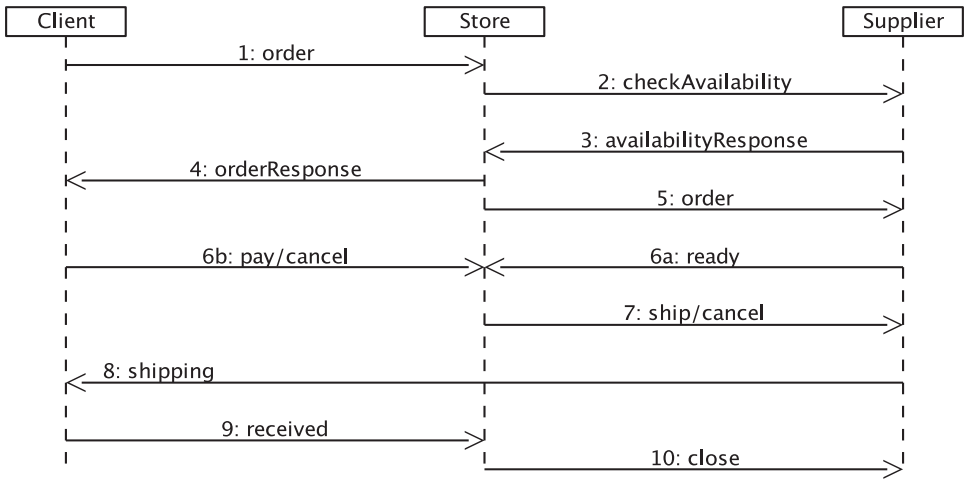
The paper is organised as follows. First, we introduce an example business protocol and its role-specific views represented by processes (Section 2). The architecture of the system is described in Section 3. In Section 4, we present a performance evaluation of the architecture and show that it imposes only minimal overhead. Section 5 discusses related work. In Section 6, we draw some conclusions.

2 Example

2.1 Purchase order protocol

The example protocol represents the placing of an order, payment and shipping of a product. The protocol involves three partners: the client, the store and the supplier. The store acts as a broker between the client and the supplier. The protocol is defined using the Message Sequence Chart of Figure 2. The protocol is similar to the Web Services Interoperability (WS-I) sample application (WSI, 2003).

Figure 2 Global view of the purchase order protocol. Message sequence chart listing all messages exchanged by the three roles involved in the protocol



The protocol consists of two main phases. The first phase of the protocol is used to determine the availability of the product. This phase starts with the client sending an order request (1) to the broker. The message contains information on the product, which is ordered. Since the store maintains no stock, it checks the availability of the product at the supplier (2). Triggered by the *checkAvailability* message, the supplier checks its stock and responds with a positive or negative answer according to the availability of the product (3). The store forwards this information to the client (4). A negative answer to the client means that the product is not available and the protocol terminates. A positive answer indicates the product is available and the protocol enters into phase two.

In phase two, the client pays for the order and the supplier ships the product to the client after receiving a confirmation by the store. The communication concerning the preparation of the shipment consists of two messages. The store orders the product from the supplier (5) and when the supplier has prepared the order for shipping it notifies the store (6a). The payment is accomplished with a single message from the client to the store (6b). Instead of paying, the client also has the possibility of cancelling the order.

Thus, after sending the order request to the supplier (5), the store waits for the notification from the supplier (6a) and for the client to pay or to cancel the order (6b). In the case of a cancellation, the store needs to notify the supplier about the client's decision by sending a *cancel* message (7). After cancelling, the protocol ends. Otherwise, the store instructs the supplier to ship the goods to the client (7). Consequently, the supplier

notifies the client that the goods are on their way (8). When the client has received the product, it notifies the store (9), which in turn completes the protocol by sending a final *close* message to the supplier (10).

2.2 Processes for the protocol participants

The control flow abstractions provided by a business process modelling language are used to capture temporal constraints between its activities. In the context of business protocols, the control flow dependencies between activities can be used to model which set of messages must have been exchanged before a certain message is allowed to be transferred. Figures 3, 4 and 5 show the processes involved in the Purchase Order (PO) protocol in WS-BPEL (OASIS, 2005).[1] An additional graphical view of the public processes using the BPMN (BPMI, 2004) syntax helps to understand the textual version. For the client and supplier the graphical version also shows some activities of an imaginary private process (dashed boxes), which is a specialisation of the corresponding public process (van der Aalst and Weske, 2001).

Figure 6 unifies the message sequence chart of the PO protocol and the public processes of the protocol participants. In the graphical processes, solid arrows represent control flow dependencies while dashed arrows represent message exchanges. Control flow dependencies can denote exclusive paths (XOR) or have a condition attached.

In all processes, a top-level sequence orders all the message exchanges. In the client process, the first two messages are mapped directly to activities. The possibility to send either a *pay* or a *cancel* message is modelled by a switch which branches based on the black-box function `clientWantsToCancel()`. After cancelling, the protocol ends. Otherwise, after paying, the *shipping* and *received* messages are exchanged in sequence.

Figure 3 Public process of the client. The graphical view additionally shows activities from an imaginary private processs (dashed boxes).

```
<process name="client">
 <sequence>
  <invoke store.order/>
  <receive store.orderResponse/>
  <switch>
   <case condition=
         "clientWantsToCancel()">
    <invoke store.cancel/>
   </case>
   <otherwise>
    <sequence>
     <invoke store.pay/>
     <receive supplier.shipping/>
     <invoke store.received/>
    </sequence>
   </otherwise>
  </switch>
 </sequence>
</process>
```

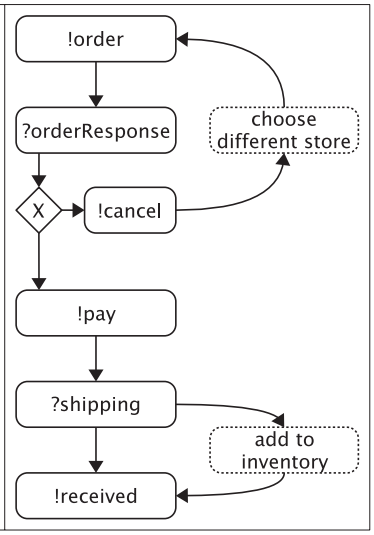
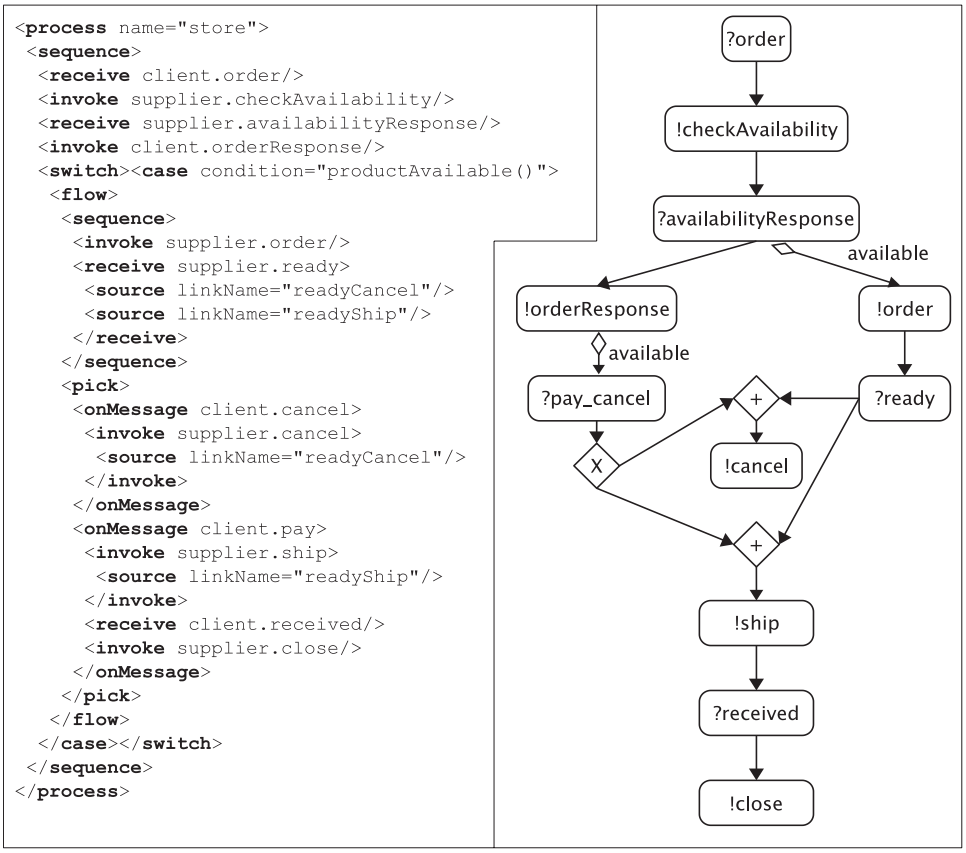
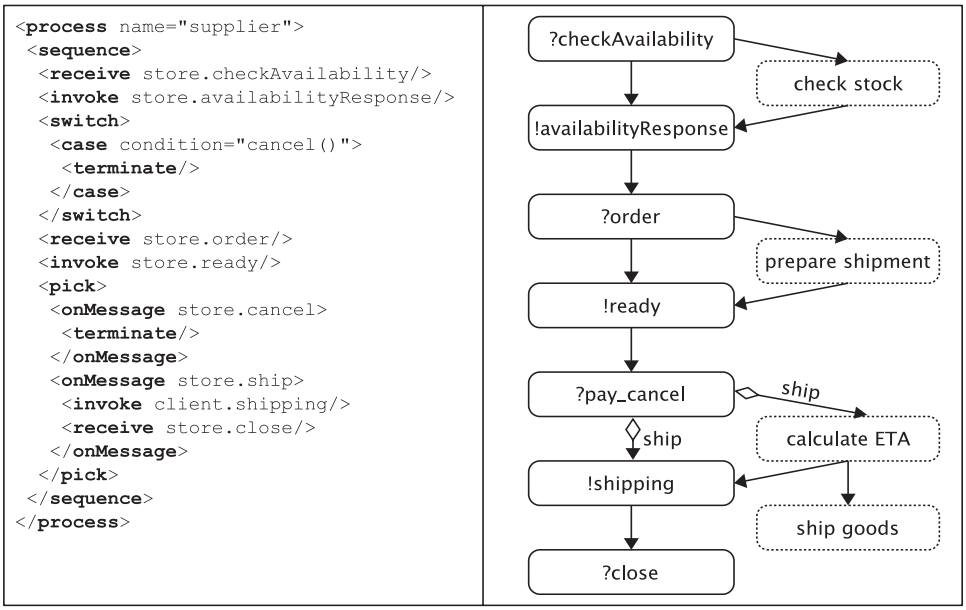


Figure 4 Public process of the store



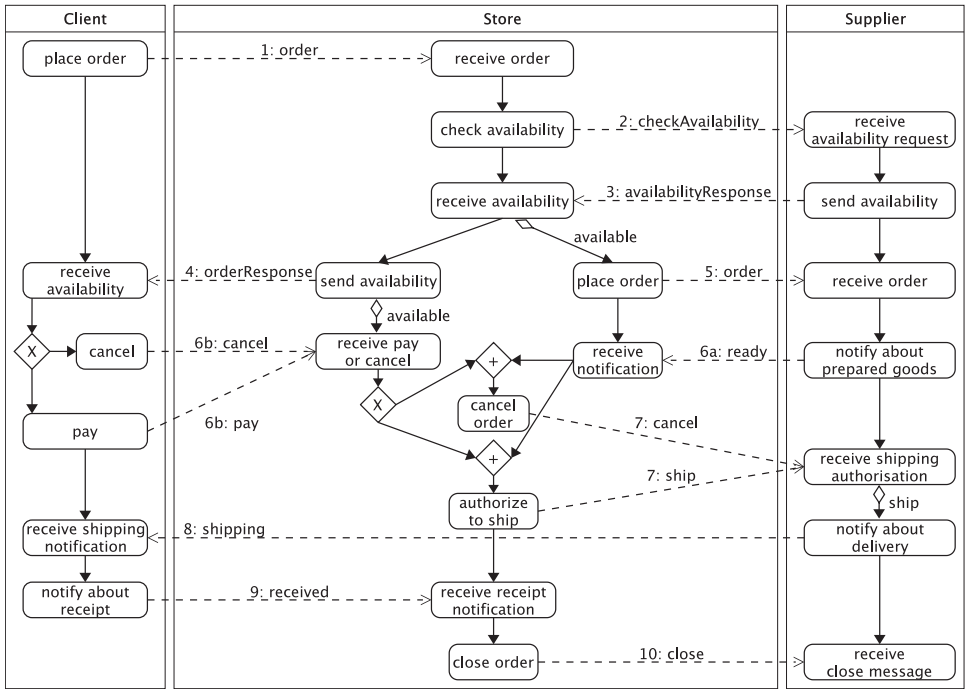
```
<process name="store">
 <sequence>
  <receive client.order/>
  <invoke supplier.checkAvailability/>
  <receive supplier.availabilityResponse/>
  <invoke client.orderResponse/>
  <switch><case condition="productAvailable()">
   <flow>
    <sequence>
     <invoke supplier.order/>
     <receive supplier.ready>
      <source linkName="readyCancel"/>
      <source linkName="readyShip"/>
     </receive>
    </sequence>
    <pick>
     <onMessage client.cancel>
      <invoke supplier.cancel>
       <source linkName="readyCancel"/>
      </invoke>
     </onMessage>
     <onMessage client.pay>
      <invoke supplier.ship>
       <source linkName="readyShip"/>
      </invoke>
      <receive client.received/>
      <invoke supplier.close/>
     </onMessage>
    </pick>
   </flow>
  </case></switch>
 </sequence>
</process>
```


Figure 5 Public process of the supplier. The graphical view additionally shows activities from an imaginary private process (dashed boxes).



```
<process name="supplier">
 <sequence>
  <receive store.checkAvailability/>
  <invoke store.availabilityResponse/>
  <switch>
   <case condition="cancel()">
    <terminate/>
   </case>
  </switch>
  <receive store.order/>
  <invoke store.ready/>
  <pick>
   <onMessage store.cancel>
    <terminate/>
   </onMessage>
   <onMessage store.ship>
    <invoke client.shipping/>
    <receive store.close/>
   </onMessage>
  </pick>
 </sequence>
</process>
```

Figure 6 Unification of the PO protocol and the corresponding public processes



In the store process, after receiving the availability from the supplier, the store forwards the information to the client. At the same time – and only if the product was available – the store places the definitive order with the supplier and waits for the *ready* message, and in parallel (using a `flow`) waits for the client to *pay* or *cancel* using a `pick`. The `pick` corresponds to the `switch` in the client process.

After reporting on the availability of the product, the store decides whether it should terminate depending on the reported availability. When the order has been prepared *(ready* message), the store uses a `pick` to receive the decision of the store whether to cancel the order or allow shipping of the goods.

3 System architecture

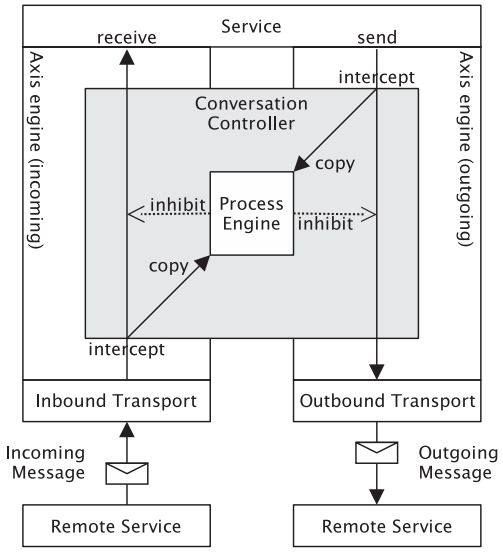
3.1 Overview

Our main goal is to enforce business protocols in a transparent manner. To do this, we introduce a conversation controller component that intercepts messages to and from the WS. The basic idea is to block messages that do not comply with the business protocol.

Figure 7 shows the architecture of the system. We assume that the WS is implemented using the Apache Axis message processing framework (Apache Software Foundation 2005b).[2] The controller is implemented as a *message handler* for Axis. As we are going to describe, such a handler can transparently intercept, manipulate, and even reject messages, which are exchanged by a service. The conversation controller contains a process engine which tracks the conversations in which the WS is involved and decides

on the validity of messages, which pass through the controller. The following shows how Axis, the service implementation, the conversation controller and the process engine work together.

Figure 7 Transparent integration of a process-based conversation controller in the WS messaging stack



3.2 Axis architecture

The main concept in the Axis architecture (Apache Software Foundation, 2005a) is the *message handler*, which can read and arbitrarily change a message. It can also abort the further processing of the message by reporting an error to Axis. Messages directed at a service, as well as messages sent by a service, are processed by the *Axis engine* (Figure 7). The engine can be viewed as two chains of handlers, one for incoming and one for outgoing messages. An incoming message is picked up by the *inbound transport* and injected into the Axis engine. An outgoing message is generated by the service and passed to the engine. The message is then processed in sequence by the applicable handlers and either reaches the service implementation object or the *outbound transport*.

3.3 Integration of the conversation controller

When a service sends messages, it is important that these are also validated and tracked by the conversation controller in exactly the same way incoming messages are checked. Incoming messages are often a reply to previously sent messages, the validity of which can only be determined if the controller knows the messages that triggered them. In our example protocol, the very first message is an outgoing message from the client. In order to correlate later answers (*e.g.*, the acknowledgment received from the store), the process engine needs to extract the correlation information from the message. Also, the *shipping* message can only be processed if the client has sent a *pay* message before.

Since the processing of incoming and outgoing messages with Axis is symmetric, the conversation controller can intercept outgoing messages in the same way as incoming ones. To do so, the conversation controller is added to the handler chains for both incoming and outgoing messages (Figure 7). When it is invoked, the controller sends a copy of the message to the process engine and indicates whether the message is incoming or outgoing. Then it waits for the engine to accept or reject the message. If the message is accepted, the conversation controller returns control to Axis and the message eventually reaches the service (inbound message) or the outbound transport (outbound message). In case the message is rejected, the conversation controller can act in different ways. It can just discard the message and ignore it, or it can reject it by notifying the sender of the message. In our implementation, we chose to reject the message. By discarding the message, the service is protected from corrupt messages, but the client will not notice the protocol violation and will keep resending incorrect messages. The explicit rejection thus helps detect errors in remote applications. To do so, the conversation controller throws an exception, which causes Axis to stop further processing of the message and to return a fault to the originator of the message (the local service for outbound messages or a remote service for incoming ones).

3.4 Managing the conversation life cycle

Every conversation is tracked by process instances running in the process engines located at each participant. Thus, we do not rely on a centralised coordinator, but use a distributed approach to enforce the protocol. At each participant, incoming messages are delivered to the corresponding process instance in order to update its state. The life cycle of a process instance corresponds to the life cycle of the conversation as seen from the respective partner.

Process creation is accomplished by *instantiating activities* (WS-BPEL: `receive` or `onMessage` with `createInstance="yes"`). When a message for such an activity arrives, the containing process is instantiated before the receipt. Each process must have at least one instantiating activity among the initial activities in the control flow. For each of the processes for the example protocol, the first activity is an instantiating activity.

A process instance can terminate in two ways. It terminates implicitly if there are no more active activities because, according to the protocol, there are no more messages, which may be sent. The process may also be terminated explicitly by a special activity in the process (`terminate` in WS-BPEL). This happens when a condition is met for which the protocol explicitly states termination.

3.5 Processes for tracking conversations

The process representing the view of a protocol participant contains both sending and receiving activities (Figures 3, 4 and 5). In contrast, the conversation controller only intercepts messages, *i.e.*, it never sends messages but rather receives a copy of messages sent by the service. Thus, the process used to model the protocol cannot be directly executed by the process engine inside the controller. This section shows how to derive the process for the conversation controller from a public process. In general, a part of a

process, which cannot be executed directly, needs to be replaced by its dual construct. We illustrate this with two examples of WS-BPEL activities in the client process: `invoke` and `switch`.

An `invoke` means the service will send a message which the conversation controller should intercept, *i.e.*, the process engine will receive a copy of it. For example, the client of the PO protocol sends the *order* message with `<invoke store.order/>`. The corresponding activity to intercept this message is `<receive client.order/>` which is also used by the store to receive the message.

The client process uses a `switch` activity to model alternative paths in the protocol (Figure 3). However, the conversation controller cannot execute a `switch` because it should only detect the branching decisions made by the service. The dual construct is the same `pick` activity as in the process of the store (Figure 4). This activity allows the store to receive either a *cancel* or an *order* message from the client. Such a `pick` can also be used in the conversation controller at the client to detect which message the client sends to the store. This way, the controller may branch the execution of the process based on the client's decision.

3.6 Message validation in the process engine

In every state of the conversation, certain `receive` activities in the process are active. They specify what message type they accept (`portType` and `operation`) and whether it is an incoming or outgoing message `(partnerLink)`. They also specify how messages should be correlated with process instances (correlation sets in WS-BPEL).

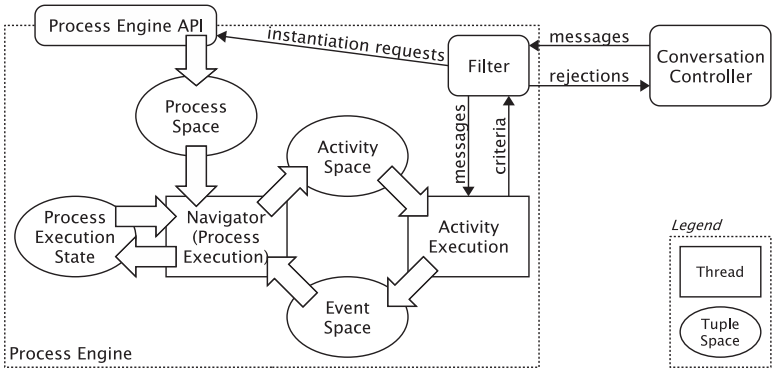
Every message, which is passed to the process engine, is correlated with the process instances. If there is an instance waiting for that message, the engine accepts it and delivers it to the instance. If the message initiates a conversation, the process engine creates a process instance for this conversation before accepting the message. When a process instance receives a message, its state is updated accordingly, thus tracking the progress of the conversation. This activates new activities in the process. These accept messages which are valid in the new state of the conversation.

If the process engine cannot deliver the message to any of the process instances, the message is rejected. The acceptance or rejection of each message is signalled to the conversation controller, which acts accordingly.

3.7 Process engine

We briefly describe how processes are executed by introducing the architecture of JOpera (Pautasso, 2004) (Figure 8), a general purpose process engine which was embedded into the conversation controller. The execution of a process begins with a request sent through the API of the engine or generated by an arriving message (see below). Such requests are queued into the *process execution request space* (or process space). These requests are handled by the navigator, which 1) creates a new process instance into the *process execution state space* and 2) begins with the actual enactment of the process. To do so, the navigator uses the current state of the execution of a process to determine which activities should be invoked next, based on the control and data flow dependencies that are triggered by the completion of the previous activities. Once the navigator determines that a certain activity is ready to be invoked, the corresponding tuple is stored in the *activity execution request space* (or activity space).

Figure 8 Logical architecture of the JOpera process execution engine



The invocation of the activities is managed by the *activity execution* component. When a `receive` activity is executed, it is registered in the *filter* component together with the restrictions it imposes on the message to be received. In the case a process is to be instantiated by a message, the corresponding criteria are registered together with the instruction to create a process instance when a message is matched.

Every message coming from the conversation controller is checked by the filter against all the criteria specified by `receive` activities. If the message matches the criteria of an activity, it is assigned to the activity, which is then marked as finished, and the navigator is notified through the event space. The activity completion events are consumed by the navigator in order to update the state of the corresponding process instance and carry on with its execution. If an arriving message does not match any of the registered criteria, the conversation controller is instructed to reject the message.

4 System evaluation

In order to measure the impact of the conversation controller on the performance of a WS, we have performed two different experiments. First, we used a very simple service to measure the overhead of the different components in the system. Second, we used the protocol of Section 2.1 to compare the conversation controller with a custom protocol enforcement layer written in plain Java.

4.1 Setup

The measurements were done over the internet between three sites: McGill University, Canada, Universidad Politecnica de Madrid, Spain, and ETH Zurich, Switzerland. McGill and ETH were 18 hops apart, ETH and Madrid 13 hops and Madrid and McGill 17 hops.

The machine at McGill was equipped with an Intel Pentium 4 processor with Hyper-Threading enabled running Linux 2.4.20-8smp at 3GHz and with 1GiB of main memory. At ETH Zurich, the machines were powered by two AMD Athlon MP 1600+ processors running Linux 2.6.8-1.521smp at 1.4GHz and with 1GiB of main memory. In Madrid, we used an AMD Athlon MP 1800+ machine running Linux 2.4.20-8smp at 1.5GHz and with 512MiB of main memory.

All SOAP processing was done with Apache Axis version 1.2alpha. All sites used Sun's Java HotSpot VM 1.4.

4.2 System overhead

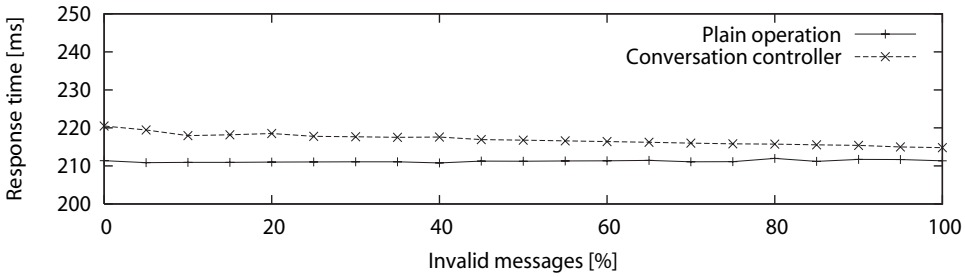
This experiment measures the overhead of the conversation controller, which occurs every time a message is intercepted. It uses a service with a single operation, which returns a short string. Although the conversation is only one message long, the controller can be used to filter invalid messages that do not satisfy data format constraints.

The service is deployed with and without conversation controller. When the conversation controller is used, a single process instance continuously checks the operation invocations. Technically speaking, this maps to a conversation with an unlimited number of invocations of one operation. The overhead of instantiating a process is not measured. The experiment uses different ratios of valid to invalid messages. Valid messages are SOAP-RPC requests for the service operation, while invalid messages are SOAP-RPC requests for a non-existing operation. A single-threaded client calls the two operations at the appropriate ratio 1000 times in total.

The WS was running at ETH and the client at McGill. The response time for each operation invocation and the throughput were measured.

Figure 9 shows the results. The overhead of the conversation controller ranges from 1.6% to 4.3% depending on the ratio of valid to invalid messages.

Figure 9 Average response time of the simple WS operation for different amounts of invalid messages



4.3 Detailed analysis of the overhead

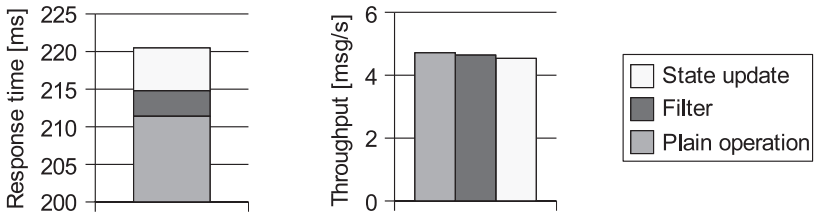
For the plain operation without conversation controller, the response time and throughput remain constant regardless of the ratio of valid to invalid messages. This shows that the time Axis takes to verify a message is constant. Axis only needs to check if the service object implements the requested method and whether it is allowed to be called. Also, the results show that the invocation of the operation and the generation of a fault take the same time.

When the conversation controller is employed, the response time for valid messages is larger than for invalid messages. The reason is the following. Every message is filtered by the process engine. Invalid messages are rejected immediately. Valid messages cause the state of the corresponding process instance to be updated, which imposes an additional overhead.

Figure 10 (left) shows the breakdown of the response time. The time is split into the portions contributed by the three steps discussed above. 'Plain operation' (211.39 ms) is the response time (including network delay, SOAP processing *etc.*) measured without conversation controller. The 'filter' part (3.4 ms) is the additional time it takes to correlate the message, regardless of its validity. Updating the state of a process instance adds 'state update' time (5.71 ms) for valid messages. The total overhead for a valid message is 4.3%.

Figure 10 (right) shows the throughput for the plain operation (4.72 messages/s), for the case where all messages are invalid and are rejected after filtering (4.64 messages/s) and for the case where all messages are valid and cause a state update (4.54 messages/s).

Figure 10 Overhead of the conversation controller. Processing delay per layer (left) and throughout (right) for the simple operation



4.4 Comparison with handwritten controller

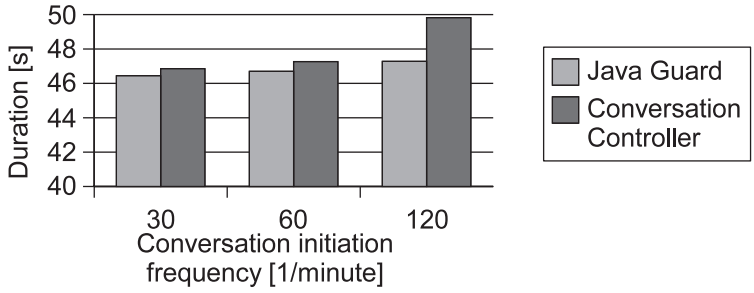
We have implemented the purchase order example protocol detailed in Section 2.1 with Java. To simulate real PO participants who access back-end systems, each of them always waits five seconds before sending a message. Thus, a complete run of the protocol takes at least 45 seconds.

This basic implementation has no special provisions for checking the protocol compliance of clients. The protocol enforcement is accomplished in two different ways. The first approach wraps each operation with a *Java guard*. For this, the Java classes of the protocol participants are sub-classed. Every published method is overridden to accept or reject an invocation based on previous operation invocations. For valid messages, the overridden method is invoked. The second approach uses the conversation controller as discussed in Section 3.3. The controller contains the process definitions describing the view of the business protocol from the guarded service.

We measured the performance of the two approaches. Each role of the protocol was running on a machine at one of the sites described above. The client was running at Madrid, the store at ETH and the supplier at McGill. A program on the client machine was initiating conversations at different frequencies by sending start-messages to the client service.

Figure 11 shows the duration of the purchase order protocol as seen from the store. At 30 conversation initiations per minute the duration is almost the same for the Java guard (46.45s) and conversation controller (46.86s) (+0.88%). At 60 conversations per minute, the durations for the protocol with Java guard (46.70s) and with controller (47.27s) are still similar (+1.21%). At 120 per minute the overhead of the controller grows to 5.37% (49.82s versus 47.28s).

Figure 11 Duration of the purchase order protocol as seen from the store. Conversations were initiated with different frequencies



4.5 Discussion

The measurements show that the overhead of the conversation controller is small when used in a realistic setting. Even for very short operations, the overhead is acceptable. Considering real operations, which last more than one second, the overhead is less than 1% for a single operation invocation. The purchase order experiment shows that for a realistic workload of 30 to 90 concurrent conversations, the overhead is from 1% to 5%.

The results are encouraging, especially because JOpera is a general purpose process support system, and hence, the performance of the system could be improved by tailoring it to conversation tracking. Currently, JOpera employs a single navigator thread for executing processes, which becomes a bottleneck when tracking many conversations in parallel. By making the navigation multi-threaded, the navigation throughput of the engine could be significantly increased. Another time-consuming step is the filtering of messages. By employing an optimised XML message filter such as YFilter (Diao *et al.*, 2003) the overhead of the message correlation could be lowered.

5 Related work

This paper relies on previous work on the theory of protocol validation. For example, (Benatallah *et al.*, 2004) discuss the compatibility of different services in terms of their protocol specifications. Several operators for the manipulation of protocols are developed. Using these operators, *compatibility* and *replaceability* levels of services are defined. For the public processes used with our conversation controller, we assume that a similar validation has been applied to make sure they comply with the global protocol. Public processes can also be derived from a global model of the protocol as described by (van der Aalst and Weske, 2001). Private processes can be created by sub-classing public processes, so that the correctness of the overall interaction can be guaranteed. In our view, services are not implemented using processes and, therefore, cannot benefit from such an approach.

Less work has been published regarding the validation of conversations at run-time. Similar to our approach, separating the protocol enforcement from the actual service implementation (Molina-Jimenez *et al.*, 2004) describe how to convert conventional contracts into *executable contracts* which are represented by Finite State Machines

(FSM). These FSMs can be executed by a middleware to verify the behaviour of the contracting parties. Kuno and Lemon (2001) describe a conversation controller, which acts as a proxy to enable services to interact even if they do not support the same protocols. The authors discuss a prototype, which uses HP's Conversation Definition Language (CDL) to describe the protocol. Unfortunately, the overhead incurred on the communication when using this approach is unknown. Pavel *et al.* (2005) propose a component model where software components are associated with protocols based on Symbolic Transition Systems. A basic component and its protocol are composed into a *controlled component*, which incorporates a mechanism to impose the specified protocol on the communication with the software component. In this approach, messages are also intercepted and validated based on the current state of the conversation.

6 Conclusion

In this paper, we have proposed to enforce business protocols at run-time using a process-based approach. We have presented an architecture featuring a distributed conversation controller, which is added transparently to the environment of a WS without having to change a possibly existing application. The conversation controller intercepts all messages sent or received by the service, updates its internal representation of the conversation state and decides on the validity of the message. Incorrect messages may thus be rejected without ever reaching the service implementation. Our approach lets the developer of the WS focus on its business logic since the business protocol is enforced from its specification. Our measurements show that the performance overhead of the proposed architecture is low even though we used a general purpose process support system to track the state of conversations.

References

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004) *Web Services – Concepts, Architectures and Applications*, Springer.

Apache Software Foundation (2005a) *Axis Architecture Guide, Version 1.2*, http://ws.apache.org/axis/java/architecture-guide.html.

Apache Software Foundation (2005b) *Axis version 1.2*, http://ws.apache.org/axis/.

Benatallah, B., Casati, F. and Toumani, F. (2004) 'Analysis and management of Web Service protocols', *Proceedings of the 23rd International Conference on Conceptual Modeling*, Shanghai, China, Vol. 3288 of *LNCS.*

BPMI (2004) *Business Process Modeling Notation (BPMN), Version 1.0*, http://www.bpmn.org/Documents/BPMN V1-0 May 3 2004.pdf.

Bussler, C. (2002) 'Public process inheritance for business-to-business integration', *Proceedings of the 3rd International Workshop on Technologies for E-Services (TES 2002)*, Hong Kong, China.

Chiu, D.K., Cheung, S., Till, S., Karlapalem, K., Li, Q. and Kafeza, E. (2004) 'Workflow view driven cross-organizational interoperability in a web service environment', *Information Technology and Management.*

Curbera, F., Khalaf, R., Mukhi, N., Tai, S. and Weerawarana, S. (2003) 'The next step in Web Services', *Communications of the ACM.*

Diao, Y., Altinel, M., Franklin, M.J., Zhang, H. and Fischer, P.M. (2003) 'Path sharing and predicate evaluation for high-performance XML filtering', *ACM Trans. Database System.*

Kindler, E., Martens, A. and Reisig, W. (2000) 'Inter-operability of workflow applications: local criteria for global soundness', *Business Process Management*.

Kuno, H. and Lemon, M. (2001) 'A lightweight dynamic conversation controller for e-services', *Technical Report*, Hewlett-Packard Laboratories.

Leymann, F., Roller, D. and Schmidt, M-T. (2002) 'Web services and business process management', *IBM Systems Journal*, Vol. 41, No. 2, pp.198–211.

Molina-Jimenez, C., Shrivastava, S., Solaiman, E. and Warne, J. (2004) 'Run-time monitoring and enforcement of electronic contracts', *Electronic Commerce Research and Applications.*

OASIS (2005) *Web Services Business Process Execution Language Version 2.0 (Working Draft)*, OASIS, http://www. oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

Pautasso, C. (2004) 'A flexible system for visual service composition', *PhD Thesis*, ETH Dissertation Nr. 15608, http://www.iks.inf.ethz.ch/publications/cp_diss.html.

Pavel, S., Noye, J., Poizat, P. and Royer, J-C. (2005) 'A Java implementation of a component model with explicit symbolic protocols', *Proceedings of the 4th International Workshop on Software Composition (SC 2005)*, Edinburgh, Scotland.

Schulz, K.A. and Orlowska, M.E. (2004) 'Facilitating cross-organisational workflows with a workflow view approach', *Data and Knowledge Engineering.*

van der Aalst, W.M.P. and Weske, M. (2001) 'The p2p approach to interorganizational workflows', *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CaiSE'01)*, Interlaken, Switzerland.

W3C (2002) *Web Services Conversation Language (WSCL) 1.0*, http://www.w3.org/TR/wscl10/.

Weerawarana, S., Curbera, F., Leymann, F., Storey, T. and Ferguson, D.F. (2005) *Web Services Platform Architecture*, Prentice Hall PTR.

WSI (2003) *Supply Chain Management Sample Application*, http://www.ws-i.org/deliverables/ workinggroup.aspx?wg=sampleapps.

Notes

1 To enhance the readability of our examples we compact the syntax of WS-BPEL as follows. The `invoke` activity (which sends a message to a service) and the `receive` activity (which receives a message from a service) have a `portType` and an operation parameter which determine what message type is exchanged. The `partnerLink` parameter specifies the partner with which the message is exchanged. We omit the `portType` and write the other two parameters as `partnerLink.operation`. References to variables are omitted as well.

2 This assumption does not limit the applicability of our approach as several other web service toolkits provide similar message interception mechanisms.