Service Interaction Patterns

Alistair Barros[1], Marlon Dumas[2], and Arthur H.M. ter Hofstede[2]

[1] SAP Research Centre, Brisbane, Australia
alistair.barros@sap.com
[2] Queensland University of Technology, Australia
{m.dumas, a.terhofstede}@qut.edu.au

Abstract. With increased sophistication and standardization of modeling languages and execution platforms supporting business process management (BPM) across traditional boundaries, has come the need for consolidated insights into their exploitation from a business perspective. Key technology developments in BPM bear this out, with several web services-related initiatives investing significant effort in the collection of compelling use cases to heighten the exploitation of BPM in multi-party collaborative environments. In this setting, we present a collection of patterns of service interactions which allow emerging web services functionality, especially that pertaining to choreography and orchestration, to be benchmarked against abstracted forms of representative scenarios. Beyond bilateral interactions, these patterns cover multilateral, competing, atomic and causally related interactions. Issues related to the implementation of these patterns using established and emerging web services standards, most notably BPEL, are discussed.

1 Introduction

Process modeling languages have emerged as a key instrument for achieving integration of business applications both within and across organizations in a service-oriented architecture (SOA) setting. This trend is reflected in a number of standardization initiatives such as the set of WS-* Specifications [11], OMG's Enterprise Collaboration Architecture[1] and RosettaNet[2], all of which position processes at the highest level of abstraction. Process modeling languages provide an abstract means of specifying complex sequences of execution steps, leaving lower layers to deal with details like software interfacing, quality of messaging and transport protocol binding. From the SOA prism, process steps result in interactions with (web) services that encapsulate the business logic associated to the step. Processes that rely on services to realize process steps can themselves be deployed as services, a practice known as *process-based service composition*.

Through different insights from various initiatives over the last few years, different aspects of process-based service composition have evolved. In partic-

[1] http://www.omg.org/technology/documents/formal/edoc.htm
[2] http://www.rosettanet.org

W.M.P. van der Aalst et al. (Eds.): BPM 2005, LNCS 3649, pp. 302–318, 2005.

ular, the developments of the Business Process Execution Language (BPEL)[3] and W3C's Web Services Choreography Definition Language (WS-CDL)[4], have been accompanied by requirements and use cases gathering. However these have largely steered towards technical concepts and implementation concerns, with documented use cases and examples reflecting little more than simple processes involving basic "buyer-supplier-shipper" interactions.

For service composition technology to progress further, more requirements gathering is needed to shed light into the nature of *service interactions* in collaborative business processes. In particular, it must be considered that there is often a large number of parties in such collaborative processes and thus the nature of interactions may be multilateral rather than bilateral. Furthermore, the assumption of strict synchronization of all canvassed responses breaks down due to the independence of the parties. More realistically, responses are accepted as they arrive or a minimum number is required for an interaction to be successful. Another crucial feature is that not all service providers have comparative advantage and collaborate. Not untypically, they compete. Hence, canvassed requests to competing service providers may require exclusivity – e.g the first response is accepted and the rest ignored. Finally, not all interactions follow a requestor-respondent-requestor structure. Instead, a sender may redirect interactions to nominated delegates and services may outsource requests choosing to "stay in the loop" and partially observe follow-ups. More generally, it may only be possibly to determine the order of interactions at runtime given the message contents.

This paper aims at contributing to this requirements gathering activity by proposing a set of *service interaction patterns*. Patterns have proved invaluable in the reuse of requirements, design and programming knowledge. They were traditionally the province of software design, but have recently emerged in the BPM field [1]. The collected service interaction patterns apply primarily to the service composition layer (orchestration, and choreography) but also to lower layers (e.g. message typing and addressing). They have been derived and extrapolated from insights into real-scale B2B transaction processing, use cases gathered by standardization committees (e.g. BPEL and WS-CDL), generic scenarios identified in industry standards (e.g. RosettaNet Partner Interface Protocols), and case studies reported in the literature. It is not claimed that the proposed set of patterns is complete: the aim is rather to consolidate recurrent scenarios and abstract them in a way that provides reusable knowledge. Furthermore, the patterns allow the assessment of emerging web services standards. Specifically, we use the patterns to analyze the scope and capabilities of BPEL and to some extent of related specifications such as WSDL and WS-Addressing (WS-A) [11].

The proposed patterns are classified according to the following dimensions:

- The maximum number of parties involved in an exchange, which may be either two (*bilateral interactions*, covering both *one-way* and *two-way* interactions) or unbounded (*multilateral interactions*).

[3] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`. In this paper, we use the acronym BPEL to refer to WS-BPEL version 2.0.

[4] `http://www.w3.org/TR/ws-cdl-10`

- The maximum number of exchanges between two parties involved in a given interaction, which may be either two (in which case we use the term *single-transmission interactions*) or unbounded (*multi-transmission interactions*).
- In the case of two-way interactions (or aggregations thereof) whether the receiver of the "response" is necessarily the same as the sender of the "request" (*round-trip interactions*) or not (*routed interactions*).

Based on these dimensions, we identify four groups of patterns. The first one encompasses single-transmission bilateral interaction patterns. These correspond to elementary interactions where a party sends (receives) a message, and as a result expects a reply (sends a reply). This group covers one-way and round-trip bilateral interactions but not routed interactions which are covered in a separate group. The second group of patterns stays in the scope of single-transmission non-routed patterns, but deals with multilateral interactions. In this case, a party may send or receive multiple messages but as part of different interaction threads dedicated to different parties. The third group is dedicated to multi-transmission (non-routed) interactions, where a party sends (receives) more than one message to (from) the same party. The final group is dedicated to routed interactions.

The proposed patterns may be composed through operators expressing flow dependencies such as sequence, choice, and synchronization. In this paper however, we do not deal with patterns composition. Also, it is not in the scope of the proposed patterns to capture internal steps performed by a service that do not directly contribute to nor directly result from interactions. Also, we abstract from data representation and manipulation issues as these deserve a separate elaboration. For the same reason, the patterns do not cover security issues.

The structure of the paper follows the groups of patterns outlined above. For space reasons, we omit the first group which comprises three well-known patterns (send, receive and send/receive) as detailed in [3]. Thus the next section starts directly with Pattern 4.

2 Single-Transmission Multilateral Interaction Patterns

Pattern 4: *Racing incoming messages.*

Description. A party expects to receive one among a set of messages. These messages may be structurally different (i.e. different types) and may come from different categories of partners. The way a message is processed depends on its type and/or the category of partner from which it comes.

Example. A manufacturing process involves remote subcontractors and uses a pull-strategy to streamline its operations. Each step in the manufacturing process is undertaken by a subcontractor. A subcontractor signals intention to execute a step when it becomes available through a request. At the same time, progress is monitored by a quality assurance service. The service randomly issues quality check requests in addition to the pre-established quality checkpoints in the process. When a quality check request arrives, it is processed in full before processing any new quality check request or subcontractor intention. Similarly,

when a subcontractor intention arrives, it is processed in full before processing any other check request or subcontractor intention. Thus, there are points in the process where quality checks and subcontractor intentions compete.

Issues/design choices.

- The incoming messages may be of different types.
- The processing that follows the message consumption (which we term the *continuation*) may be different depending on the consumed message.
- When one of the expected messages is received, the corresponding continuation is triggered. The remaining messages may or may not need to be discarded.
- Depending on the underlying communication infrastructure, several of the expected messages may be simultaneously available for consumption. In this case, two approaches may be adopted: (i) let the system make a non-deterministic choice, or (ii) provide a "ranking" among the competing messages. In any case, only one message is chosen for consumption.

Solution. This pattern is directly captured by the pick activity in BPEL. The pick activity simultaneously enables the consumption of several types of message events and allows at most one message event to be consumed. Specifically, a pick activity is composed of multiple branches, each of which has a corresponding handler which acts as the trigger of the branch. Occurrences of message events are consumed by onMessage handlers. An onMessage handler is associated with a type of message, identified by a partner link and a WSDL operation. When a message of the type associated to an onMessage handler is available for consumption, a message event may occur which is immediately consumed by the handler. The pick enforces that at most one of its associated onMessage handlers will consume an event. It is also possible to associate a timer with a branch of a pick activity through an *onAlarm handler*. The corresponding branch is taken if the timeout event occurs before any of the other branches is taken.

In the current version of BPEL, it is not possible to express a ranking among the competing types of message event handlers under a given pick. Although in the concrete syntax of BPEL the handlers under a pick are ordered, this order is not significant. Hence, should there be several onMessage handlers able to consume message events when the pick activity is executed, the system may choose any of them non-deterministically. What is needed to capture the fourth issue of this pattern is a way of ranking message events so that when several of them enter into a race, the one with highest ranking is chosen.

Related pattern.

- *Deferred choice* [1]. The deferred choice pattern corresponds to a point in a process where one among a set of branches needs to be taken, but the choice is not made by the process execution engine (as in a "normal choice"). Instead, several alternatives are made available to the environment and the environment chooses one of these alternatives. The Racing Messages pattern can be seen as a specialization of the deferred choice where the choice of branch is determined by the receipt of a message.

Pattern 5: *One-to-many send.*
Description. A party sends messages to several parties. The messages all have the same type (although their contents may be different).
Synonyms. Multicast, scatter [10].
Example. A purchasing service sends a call for tender to all known trading parties that provide a given type of product or service.
Issues/design choices.

- The number of parties to whom the message is sent may or may not be known at design time. In the extreme case, it may only be known just before the interaction occurs.
- As for the one-to-one send, reliable delivery may or may not be required. In the case of reliable delivery, the individual send actions may result in faults and thus fault handling routines should be associated to each of the individual send actions. The logic of these fault handlers is application-dependent: some applications may choose to terminate the whole one-to-many send when one of the individual "send actions" fail, while others may simply record the failures that occur and proceed.

Solution. A natural approach to address this pattern is to use the One-to-one Send pattern as a basic building block. Thus, a number of one-to-one send actions are scheduled in parallel or sequentially depending on the capabilities of the underlying language. For example:

- If the number of parties is known at design time, it is possible to capture this pattern in BPEL through a parallel block (i.e. a *flow* activity) such that each thread contains a one-to-one send action with its associated fault handler. Otherwise, the individual send actions would need to be scheduled sequentially (using a *while*) thus contradicting the essence of the pattern.
- In certain proprietary extensions of BPEL, such as Oracle BPEL[5], special constructs are provided to capture the situation where an arbitrary number of executions of an activity need to be performed in parallel, such that this number is only determined when these parallel executions are started (see for example the *FlowN* construct in Oracle BPEL).[6] The pattern can be captured using such a construct.
- In WSCI and BPML[7], a construct known as "spawn" is provided to start an instance of a sub-process asynchronously. By embedding the "spawn" within a "while" loop, it is possible to start a number of "send sub-processes", each of which would be responsible for sending one of the messages and dealing with any possible fault. These sub-processes would execute in parallel and return back to the parent process upon completion through a "signal". These signals can then be gathered by a dedicated activity in the parent process.

[5] `http://www.oracle.com/technology/products/ias/bpel`
[6] A similar construct (namely *parallel foreach*) has been proposed for introduction into the BPEL standard; see Issue 147 in the list of BPEL issues available from `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.
[7] `http://www.bpmi.org`

This pattern requires a "dynamic binding by reference" mechanism [2] since in some cases the set of potential parties to which messages will be sent is not known at design/build time. Instead, the identity and location of the partners may be given as parameter, or retrieved from a local database, or from a remote service registry. In BPEL, this is achieved by treating *service endpoints references* (described in WS-A) as first-class citizens that can be associated with predefined partner links at runtime.

Related pattern.

- *Multiple instances with a priori runtime knowledge (MIRT)* (van der Aalst et al. 2003). In this pattern, several instances of a task are created and allowed to execute in parallel with synchronization occurring when all instances have completed. The number of task instances to be created is only known at runtime, just before the instantiation starts. The one-to-many send can be expressed by composition of the MIRT pattern and the one-to-one send pattern discussed above. The FlowN construct of Oracle BPEL (see discussion above) is a realization of the MIRT pattern.

Pattern 6: *One-from-many receive.*

Description. A party receives several logically related messages arising from autonomous events occurring at different parties. The arrival of messages must be *timely* so that they can be correlated as a single logical request. The interaction may complete successfully or not depending on the messages gathered.

Synonyms. Event aggregation [8], gather [10].

Example. A group buying service receives requests for buying different types of items. When a request for buying a given type of product is received, and if there are no other pending requests for this type of item, the service waits for other requests for the same type of item. If at least three requests have been received within five days, a "group request" is created and an order handling process is started. If on the other hand less than three requests are received within the five days timeframe, the requests are discarded and a fault notification is sent back to the corresponding requestors.

Issues/design choices:

- Since messages originate from autonomous parties, a mechanism is needed to determine which incoming messages should be grouped together (i.e. correlated). This correlation may be based on the content of the messages (e.g. product identifier).
- Correlation of messages should occur within a given timeframe. The receiver should avoid waiting indefinitely.
- The number of messages to be received may or may not be known at design time or run-time. Instead, after a certain condition is fulfilled, the received messages are processed without waiting for subsequent related messages (i.e. proceed when X amount of orders for a given product have been received).
- In some cases, a timeout occurs before any message is received.

Solution, The first issue implies that the payload of the messages received should contain a piece of information that determines with which other messages

it should be grouped (i.e. in which group should it be placed). At an abstract level, this can be captured through a function *Group*: *Message* → *GroupID*, which associates a "group identifier" to a message. Messages with the same group identifier are to be correlated. When a message of the expected type is received, its group ID is inspected and one of three options may be taken: (i) a new group is created for the message if no group for that group ID exists; (ii) the message is added to an existing group; (iii) the message may be discarded because the group ID is not valid (e.g. the group existed before but it is no longer accepting new messages). The latter option entails that the recipient should maintain a list of invalid group IDs (or equivalently a set of valid ones).

Because the number of messages to be received is not necessarily known in advance, it is necessary to incorporate a notion of *stop condition*. The stop condition may be expressed as a predicate over the set of messages received. The stop condition is evaluated each time a message is received. As soon as the stop condition evaluates to true, the interaction is considered to be complete. In a tender scenario, to capture that as soon as 5 bids have been received the interaction completes and subsequent bids are ignored, the corresponding stop condition would be $|R| = 5$, where R denotes the set of messages received.

A solution to this pattern should associate timers to message groups. The timer for a group is started when the group is created. A group may be created either explicitly by the service (e.g. when the service enters a given state) or by the receipt of a message which mapps to a group ID for which no corresponding group is open. In the former case, it is possible that a timeout occurs even if no message has been received.

When a timeout occurs, depending on the set of messages gathered at that point, the interaction may be considered to have succeeded or failed. For example, a tender may be considered as successful if there are at least 3 bids and at least one of them is below a given limit price. Thus, a generic solution to the pattern also needs to incorporate a notion of *success condition* which is evaluated when the interaction completes and determines whether the interaction is considered as successful or not. Again, the success condition can be expressed as a predicate over the set of messages received. In the example at hand, the success condition would be: $|R| \geq 3 \wedge \exists r \in R : Price(r) \leq limitPrice$. Note that in theory, it may happen that the stop condition evaluates to true (and thus the interaction stops), while the success condition evaluates to false, so the interaction is considered to have failed. When a group completes successfully, the set of responses gathered for that group constitute the output of the interaction.

In the "group buying" example above, the stop and success conditions are identical ("at least three requests should be received"), the timeframe is five days, groups are created when the first message for the group arrives, and group IDs are never flagged as invalid since it is always possible to process requests for a type of product whether previous groups for this type have been filled or not.

Related pattern.

- *Multiple instances with a priori runtime knowledge* (MIRT). See discussion in the "Related patterns" paragraph of the previous pattern. Note that exist-

ing realizations of the MIRT pattern, such as the FlowN construct of Oracle BPEL (see discussion above) do not support arbitrary stop and success conditions as defined above. Instead, these conditions appear as lower and upper bounds on the number of task instances that are required to complete.

Pattern 7: *One-to-many send/receive.*
Description. A party sends a request to several other parties, which may all be identical or logically related. Responses are expected within a given timeframe. However, some responses may not arrive within the timeframe and some parties may even not respond at all. The interaction may complete successfully or not depending on the set of responses gathered.
Synonyms. Scatter-gather [10,6].
Example. An insurance company outsources some aspects of its claims validation to its external search brokers. Brokers are typically small agencies and have variable demands. For efficiency, the insurance company sends search requests to all the brokers, and accepts the first three responses to undertake the search.
Issues/design choices.

- The number of parties to which messages are sent may or may not be known at design time.
- Responses need to be correlated to their corresponding request.
- The sender should avoid waiting indefinitely or "unnecessarily" for responses.
- It is possible that no response is received.
- Reliable delivery may or may not be required during sending. In the case of reliable delivery, the individual send actions may result in faults.

Solution. A solution to this pattern can be obtained by combining patterns one-to-many send and one-from-many receive through parallel composition (e.g. "flow" construct in BPEL). Since outgoing and incoming messages need to be correlated, it is necessary to include correlation data in the outgoing messages and retrieve these data from the incoming messages. BPEL provides a declarative mechanism, namely correlation sets, for correlating communication actions (e.g. correlating an invoke action with a receive action). Unfortunately, this mechanism can not be employed if the actions to be correlated are executed in different loops located in different branches of a flow activity[8], which is the case for this pattern since an *a priori* unknown number of invoke and receive actions need to be executed in an arbitrary order. Thus the correlation between the send and the receive actions implied by this pattern needs to be handled at the application level, i.e. by introducing actions that insert and extract the correlation data into/out of the incoming/outgoing messages.

The "stop condition" and the "success condition" for the one-from-many receive may involve both the set of requests (to be) sent (say RQ) and the set of responses gathered at a certain point (say RS). For example, to capture that

[8] Specifically, in BPEL the invoke and the receive actions to be correlated must be enclosed under a common scope activity such that each of these actions is executed at most once per execution of the scope.

as soon as 10 responses have been received the interaction stops and subsequent responses are ignored, the stop condition can be set to: $|RS| = 10$. Meanwhile, to ensure that at least 50% of the parties need to respond the success predicate should be set to: $|RS| = 0.5 \times |RQ|$.

In the absence of a "stop condition" (i.e. if the stop condition is always true) the pattern can be expressed by combining several elementary send and receive actions through parallel composition which may be preempted by a timeout. As discussed in the previous pattern, this would mean that the underlying language provides a mechanism for executing an a priori unknown number of activities in parallel, such as for example the "FlowN" construct in Oracle BPEL or the "spawn" construct in BPML. Such a mechanism is not present in standard BPEL and a workaround solution where the various one-to-one send/receive would be executed sequentially does not properly address the pattern.

In the case of reliable delivery, fault handling routines (BPEL fault handlers) may be attached either to each individual send actions or to the whole set of send actions. A possible fault handling routine is to record that the message in question was not delivered so that this information can be used in the stop and success conditions. This way, it is possible to express conditions such as "stop as soon as half of the parties who actually received a request have responded".

Related pattern.

- *Scatter-gather* [6]. The scatter-gather pattern is a special case of the one-to-many send/receive. The scatter-gather assumes that all parties respond in a timely manner and that all responses must be gathered. Thus it does not address issues related to timeout, stop and success conditions.
- *One-from-many receive/send.* This is the dual of the One-to-many send/receive. Its description, issues, design choices, and solution are analogue to those of the One-to-many send/receive.

3 Multi-transmission Interaction Patterns

Pattern 8: *Multi-responses.*
Description. A party X sends a request to another party Y. Subsequently, X receives any number of responses from Y until no further responses are required. The trigger of no further responses can arise from a temporal condition or message content, and can arise from either X or Y's side. Responses are no longer expected from Y after one or a combination of the following events: (i) X sends a notification to stop; (ii) a relative or absolute deadline indicated by X; (iii) an interval of inactivity during which X does not receive any response from Y; (iv) a message from Y indicating to X that no further responses will follow. From this point on, no further messages from Y will be accepted by X.
Synonyms. Streamed responses, message stream
Example. A goods deliverer provides an urgent transportation service on behalf of suppliers to customers in a city. For optimization of travel, it subscribes to a local traffic reporting service provides its destination nodes (goods dispatch

and customer locations) and obtains regular feeds on traffic bottlenecks, until it indicates that no feeds are required.

Issues/design choices.

- Party X should be capable of receiving multiple messages from party Y including ones that arrive simultaneously. The number of responses accepted will depend on a condition to be evaluated at runtime.
- As with Pattern 4, the messages may be of different types. The way each message is processed depends on its type.
- As with the One-from-many Receive pattern, a stop condition is pertinent. However, unlike the One-from-many Receive, a success condition does not apply since faults messages received by X are treated individually just as "normal" messages. It is assumed that X and Y establish an *a priori* understanding of the stop condition.
- In the case where X determines when the multi-transmission should stop, there is an interval between the moment when X decides to stop and the moment when Y becomes aware of this decision. During this interval, Y may send messages that will then be rejected by X. Hence, a mechanism should be in place for Y to know that its messages have been rejected.

Solution. As for Pattern 4, the core of this pattern can be captured in BPEL through a pick activity with a onMessage handler per type of message (whether a normal message or a fault message). To capture the fact that several messages may be accepted, the pick activity must be embedded within a "while" activity. The encoding of the stop condition depends on its nature:

- If the stop condition is based on data available at the receiver's side and/or messages' content, the stop condition can be encoded as the exit condition of the while loop (like in the One-from-many receive pattern).
- If the stop condition is an absolute or a relative deadline (with respect to the beginning of the interaction), the while activity must itself be embedded in a scope activity containing an onAlarm handler corresponding to the deadline.
- If the stop condition corresponds to a period of inactivity between responses, it can be captured as a branch in the pick activity associated with an onAlarm handler capturing the maximum duration of inactivity. If this branch is taken, the while loop is interrupted (e.g. by setting an appropriate flag).
- If the stop condition is determined by the Y, a pre-agreed type of message will signal the end of the interaction to X and thus the stop condition will be encoded as an onMessage handler corresponding to this type of message.

In the case where the stop condition is determined by X, or in the case where it is determined by Y but the underlying messaging infrastructure or interaction policies do not guarantee ordered delivery of messages, X should be able to return fault messages to Y for responses that are ignored. In BPEL, this can be done by activating a thread of control after bespoke while/scope activity, which upon receiving any of the expected types of messages from Y, returns a fault message. This additional coding is necessary because in BPEL, while it is possible to state

that a process is expecting a type of message from a given party, it is not possible to express that a process expects not to receive a given type of message and that such messages should be discarded and a fault returned to their sender.

Pattern 9: *Contingent requests.*
Description. A party X makes a request to another party Y. If X does not receive a response within a certain timeframe, X sends a request to another party Z, and so on.
Synonyms. Send with failovers.
Example. A travel agency allows contingent reservations of flights in particular situations - urgent requests and busy flight paths. Customers nominate the preference of flight carriers. In order of preference, reservations are sought in short-timeframes. If a reservation is secured, the interaction ends.
Issues/design choices.

- There is a race between receiving a response and a timer.
- After a contingency request has been issued, it may be possible that a response arrives (late) from a previous request. This means that more than one response may arrive; in all, as many responses may potentially arrive as requests have been sent. The question is when to accept a response if more than one request has been made and more than one response arrives.

Solution. The first issue is generally well-understood and in fact BPEL provides direct support for it through the pick construct containing onMessage and onAlarm handlers. For the second issue, several choices are available. One is to accept the first response even if it is late and stop outstanding requests. Another is to accept the first arriving response, trigger the end of outstanding requests, but receive any further responses that arrive (before the "contingent send" process terminates). Yet another possibility is to disallow late arrivals altogether, and receive only the response of the current request. For these choices, the pattern does not pre-dispose which prevails. In some situations accepting late responses is desirable, while in others it may cause problems of integrity in remote parties particularly if requests are non-idempotent (involving database updates and extending interactions even further with other parties).

Pattern 10: *Atomic multicast notification.*
Description. A party sends notifications to several parties such that a certain number of parties are required to accept the notification within a certain timeframe. For example, all parties or just one party are required to accept the notification. In general, the constraint for successful notification applies over a range between a minimum and maximum number.
Synonyms. Transactional notification
Examples.

- *Classical "all-or-none" atomicity.* A business venture service[9] supports the process of business license applications for small business endeavors (e.g.

[9] This example reflects the Queensland Government's SmartLicence initiative (`http://www.sd.qld.gov.au/dsdweb/htdocs/slol/`)

opening a restaurant). After the steps of obtaining and verifying application details, relevant agencies involved in the approval or registration of the application are notified. All of them must receive notification as there are inter-dependent aspects of the application leading to cross-consultation. There may also be competing applications for the same business. Therefore, all agencies should receive the notification in a timely fashion. In this example, the minimum and maximum equal the number of all agencies notified.
- *Exclusive choice.* A legal firm has automated its property conveyance process for various loan types. The process utilizes a number of search brokers who have the same level of service agreements with the firm. Each of the brokers competes for conveyance applications. Therefore, only one of the notified brokers is selected, namely the first to accept the request. The minimum and maximum both are one.

Issues/design choices.

- The set of parties to which the notification will be sent may not be known at design time nor a priori at run-time.
- Specification for the minimum and maximum bounds should be supported.
- The constraint that all parties should have received the notification, means that if any one party received the notification, all the other parties also received it. Thus, some kind of transactional support is required for this aspect of the interaction.
- Following from the above point, two steps in the interaction can be seen, both of which need to be formalized. The first send-receive establishes the intention to accept a request while the second acts of the decision following an examination of received intentions - parties are notified about whether they have been selected or not.
- The maximum number of parties required to accept the notification may be less than the number of parties that notifications were sent to. Thus, more responses than the maximum allowed may be willing to accept the notification and a *preference function* may be needed to prune some of them.

Solution. The central issue of this pattern (third issue above) clearly relates to transactional atomicity. At present, BPEL does not provide support for transactional atomicity. However, it does provide support for a related notion, known as quasi-atomicity [5] through the notion of *compensation handler*. Quasi-atomicity refers to the ability to "undo" certain parts of a process execution. Using this mechanism, the receiving parties, when they receive the initial request, may actually perform the work associated with this request. Later on during the second round, if the sender decides not to proceed with the request to a given party, then that party may compensate for the work that it had previously done. However, in between these two rounds, the effects of the initial request would be visible to other parties, thus violating the principle of atomicity underlying this pattern. Supporting atomic interactions is the aim of a dedicated WS specification known as WS-AtomicTransaction[10], which provides a realization of the

[10] `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/wsacoord.asp`

distributed two-phase commit (2PC) protocol. However, this specification has not yet matured into a standardization initiative.

4 Routing Patterns

Pattern 11: *Request with referral.*

Description. Party A sends a request to party B indicating that any follow-up should be sent to a number of other parties (P1, P2, ..., Pn) depending on the evaluation of certain conditions. By default, faults are sent to these parties, but they could alternatively be sent to another nominated party (possibly party A).

Examples.

- *Referral to single party*: As part of a purchase order processing, a supplier sends a shipment request to a transport service. Subsequently, the transport service reports shipment status (e.g. as per RosettaNet's PIP 3B1) directly to the customer who then correlates these with its initial purchase order.
- *Referral to multiple parties*: After processing its inventory re-stocking for a week, a supermarket's warehouse contacts a supplier for order and dispatch of goods, notifying it of the different transport services available (different services specialize in transport of different sorts of goods). The supplier directly interacts with these transport services regarding the scheduled dispatch times (arranged by the supermarket). Faults related to order fulfillment are sent by the supplier to the warehouse, while faults related to delivery are sent by the corresponding transport services to the warehouse.

Issues/design choices.

- Party B may or may not have prior knowledge of the identity of the other parties. The information transferred from A to B must therefore allow B to fully identify and to interact with the other parties.
- The referred parties (P1, ..., Pn) and the party nominated to process faults (if different from A) may receive messages related to interactions that they did not initiate. These messages should then be related to internal processes at these parties. Sometimes, messages received through referral trigger new process instances, while other times, they will be routed to an activity within an already running process instance. The data transferred must allow the referred parties to route the message to the correct internal process.

Solution. At the messaging level, this pattern is partially addressed by WS-A which defined (among others) two fields that can be included in SOAP message headers, namely reply-to and fault-to. Using these fields, it is possible to specify the service endpoint(s) to which replies and faults should be sent. The information allowing the referred service to correlate the incoming message with its internal processes may be transferred in one of two ways depending of the adopted *state representation style* [4]: (i) it may be encoded in the endpoint reference itself (as per the REST architectural style); or (ii) it may be encoded somewhere

else in the message (e.g. in the message body). In the supplier-shipper-customer example, the supplier passes to the transport service, a reference to the customer's procurement service endpoint. In the first style above, this endpoint reference would contain a data item (e.g. the original purchase order ID) allowing the customer to correlate the message with its internal activities, while in the second style, this data item would be encoded inside the shipment notification.

At the service composition level (specifically in BPEL), endpoint references can be manipulated as ordinary data. They can be included in the contents of a message and can be dynamically bound with partner links (e.g. the partner link defined between the transport service and the customer). In addition, BPEL offers a notion of correlation set, which corresponds to information sent along a message that is used on the receiver's end to correlate that message with its internal process instance. Correlation sets can thus be used to encode correlation-related information that it not included as part of the endpoint reference.

Related pattern.

- *Channel mobility.* Channel mobility in pi-Calculus [9] refers to the ability for a process X to pass a channel name to another process Y. Passing channel names along with requests provides a means of realizing the Request with Referral pattern. In fact, this is the way the pattern is captured in BPEL, where channels names are coded as endpoint references and correlation data.

Pattern 12: *Relayed request.*
Description. Party A makes a request to party B which delegates the request to other parties (P1, ..., Pn). Parties P1, ..., Pn then continue interactions with party A while party B observes a *view* of the interactions including faults. The interacting parties are aware of this view (as part of the condition to interact).
Example. Some supportive work of managing regulatory provisions outsourced by government agencies to external agencies fits this pattern. Party A is a client seeking some outcome pending regulation, e.g. obtaining particular land tenure. Party B is the government authority concerned with the regulation. e.g. lands department. Parties P1, ..., Pn are outsourced service providers from the government authority's regulation process, e.g. brokers who validate applications and external land management experts who can provide independent audit of applications. The government authority stipulates that interactions between the client and outsourced service providers associated with key points of processing, such as the start and end of activities, and key reports, be sent to it.
Issues/design choices.

- The delegated parties (P1, ..., Pn) may or may not have prior knowledge of the identity of the request originator, party A.
- A mechanism is needed to express party B's view of interactions between party A and the delegated parties. This may include all interactions or specific ones deemed to be of interest as indicated by the content of the messages.
- The view is defined at design time, but may be modified at run-time (party B may adjust what it needs to see depending on progress of activities).

– Party B could apply referrals for redirecting interactions or faults to other parties, however this issue is orthogonal to this pattern and is covered in Pattern 11.

Solution. This pattern, like the request with referral (pattern 11), involves indirection through delegation (party B passes party A's endpoint service reference to delegated parties for further interactions) and can be effected through WS-A or exchanged message data as previously discussed. The correlation strategies similarly apply. The comparative requirement for relayed requests is representing party B's view and enforcing it, including changing it, as interactions execute as identified through the second and third issues above.

Unfortunately, WS-A does not provide direct support for including party B in the interactions due to its lack of a "Cc field". But even if WS-A offered such Cc field, it would not cover a key requirement of the pattern: The messages passed between party A and the delegated parties would be exactly the same as what party B sees. Of course, not all messages have to be "Cc-ed" to party B, but this remains a rather limited solution since whole message, not filtered messages, are transmitted to B. It is furthermore possible that B do the filtering rather than pushing this up to the level where interactions are generated. We argue, however, that view filtering decoupled from interaction generation, is deficient since party A and the delegated parties no longer have an understanding of what they are obliged to reveal to B, as required by the pattern.

This brings us to the core issue of how to specify views such that they could be deployed and utilized as part of the interaction cycle. Simple views could be specified through a querying language like XPATH while more sophisticated ones could be supported through XQuery. Party A and the delegated parties would either have static view definitions prior to run-time or they would be passed at run-time when B establishes delegation.

For dynamically modified views, B would issue new views. These need to be coordinated with A, so that both ends of interactions are subject to the new version of the view. An obvious solution is to accompany a send in an interaction with a second send for party B, conditional upon the view filter applied to the message passed through the first sent. The two sends must be atomic.

Pattern 13: *Dynamic routing.*
Description. A request is required to be routed to several parties based on a routing condition. The routing order is flexible and more than one party can be activated to receive a request. When the parties that were issued the request have completed, the next set of parties are passed the request. Routing can be subject to dynamic conditions based on data contained in the original request or obtained in one of the intermediate steps.
Synonyms. Routing slip [6,7].
Example. After processing an order, the sales department sends a request to the finance department to process the invoicing and payment receipt for the order. This request contains a reference to the customer's procurement service and possibly also to a shipping service nominated by the customer. After arranging invoicing and payment by interacting directly with the customer, the finance

service forwards the order to the warehouse service. If the order is marked "for pick-up", the warehouse eventually sends a notification of availability for pick-up to the customer's procurement service. Otherwise, the warehouse issues a request to a shipping service which may be either the company's default shipping service, or the one originally nominated by the customer. The shipping service eventually sends a shipping notification directly to the customer.

Issues/design choices.

- The set of parties through which the request will circulate may not be known in advance and these parties may not know each other at design time.
- The specification of ordering should support service-to-role late binding, parallelism and interleaved parallel routing [1], synchronization points between parallel steps, and dynamic conditions.
- A way of providing relevant (fragments of) documents to different parties needs to be supported as well as a mechanism for controlling read-only and write access to these document (fragments).
- The update of routing should be subject to role access permissions, e.g. only a project coordinator is allowed to re-route a proposal review through work-package leaders.

Solution. The requirements for dynamic routing are outside the scope of direct support through BPEL. BPEL solutions are possible but would necessarily be ad hoc and require significant amounts of hand-crafted application code. WS-Routing[11] (a proposal not yet under standardization) can serve to implement some aspects of this pattern: Parallel routing, but not interleaved parallel routing, is possible; static, but not dynamic, conditions are supported, although this and the relevant routing role matching becomes supplementary coding for the full solution. Thus, WS-Routing can support simple dynamic orders, like those of the Routing slip pattern [6]. However, the complex dynamic routes required by our examples above, cannot currently be supported.

5 Conclusion

As service composition developments unfold in their objectives of making real-scale B2B transactions a reality and ushering in newer exploitations of service interoperability, it is striking how insufficiently guided these efforts are by well-structured requirements. We sought in this paper to address this gap by establishing a reference for service interactions. We did so by distilling insights from the literature, standardization activities, and use case scenarios, to derive a set of patterns. These patterns allow relevant technologies to be benchmarked. In this paper, we have investigated BPEL's capabilities in terms of the patterns.

BPEL directly supports *single-transmission bilateral patterns*. For *single-transmission multi-lateral patterns*, BPEL restricts the send-receives to be sequential and requires "house-keeping" code for correlation and for capturing stop

[11] `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-routing.asp`

and success conditions. We recommend more effective support for these patterns through a construct capturing parallel composition of an a priori unknown number of send-receives. Of the *multi-transmission patterns*, BPEL event handling capabilities provide support for the *multi-responses* and *contingent sends*. However, lack of sufficient transaction support significantly compromises a BPEL solution for atomic multi-cast. For the *routing patterns*, simple *request referrals* are possible by passing endpoint references and implementing indirect interactions through correlation identifiers. This also serves *request relaying*. In addition, WS-A provides some support for request referrals and relaying although this support would be more direct if a Cc field was available. *Dynamic routing* is outside the scope of BPEL but WS-Routing can serve to implement some aspects of it, though not the flexible ordering and dynamic routing conditions.

Future work will extend the patterns by further extrapolations and will consider conversation management, viz. create, cancel, undo, suspend, and resume conversations. We are also drawing on insights from the patterns to design a framework for conceptual modeling of service interactions.

Acknowledgments. The authors wish to thank Phillipa Oaks, Helen Paik and Ivana Trickovic for their input and feedback. The second author is funded by a Queensland Government "Smart State" Fellowship co-sponsored by SAP.

References

1. W. M.P. van der Aalst, A. H.M. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5-51, 2003.
2. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, architectures and applications*. Springer Verlag, 2003.
3. A. Barros, M. Dumas, and A. H.M. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. Technical Report FIT-TR-2005-02, Faculty of IT, Queensland University of Technology, 2005. See: `http://www.serviceinteraction.com`.
4. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
5. C. Hagen, and G. Alonso. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering* 26(10): 943-958, 2000.
6. G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2004.
7. A. Kumar and J.L. Zhao. Workflow Support for Electronic Commerce Applications. *Decision Support Systems* 32: 265-278, 2002.
8. D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.
9. R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
10. M. Snir and W. Gropp. *MPI: The Complete Reference*. MIT Press, 2nd edition, 1998.
11. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D.F. Ferguson (Editors). *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005.