Technical Report FIT-TR-2005-02, Faculty of IT, Queensland University of Technology, April 2005.

Service Interaction Patterns:
Towards a Reference Framework for Service-Based Business Process Interconnection

Alistair Barros
SAP Research Centre Brisbane
alistair.barros@sap.com
Marlon Dumas, Arthur ter Hofstede
Queensland University of Technology
m.dumas@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 "web services standards stack", OMG's Enterprise Collaboration Architecture[1] and RosettaNet[2], all of which position processes at the highest level of abstraction. Process modeling languages, as such, provide an abstract means of specifying the complex and long-running sequences of execution steps, leaving lower layers to deliver on the infrastructure services like software interfacing, quality of messaging service and transport/protocols. From the SOA prism, process steps result in executions of web services. At the same time, processes that rely on services to realize process steps can themselves be deployed and accessed as services, a practice known as *process-based service composition*.

Through different insights from various research, development, and standardization initiatives over the last few years, different aspects of process-based service composition have evolved. As seen through the Business Process Execution Language (WS-BPEL or BPEL)[3] initiative, process constructs have been exploited to provide a classical, workflow style composition (often called *orchestration)* of services, extended with external message interaction capability.

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

[2] http://www.rosettanet.org

[3] http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel. In this paper, we refer to the draft of the WS-BPEL 2.0 specification dated 27 February 2005.

BPEL is thus used for capturing processes of one party including its external interactions with other parties. It is the latest and most significant development succeeding previous efforts including WSFL, XLANG and BPML. For process coordination across several parties, the need to expose only those details pertinent to interactions with other processes, without revealing the detailed steps of the process, has been identified. This can be seen as a *behavioral interface* (known as *abstract process* in BPEL and *collaborative protocol profile* in ebXML), focusing on incoming and outgoing message interactions of a given process and thereby establishing an interaction contract.

More recently, through W3C's Web Services Choreography Definition Language (WS-CDL) initiative, which follows other initiatives such as WSCL and WSCI (and in part ebXML), a trends towards an explicit treatment of multi-party interaction processes (better known as *choreographies*) has emerged. The focus of choreography models, as opposed to orchestration models, is on the message exchange sequences across multiple parties, without details of internal processing. Also, in contrast to both orchestration models and behavioral interfaces, choreography models adopt a global view of interactions. A choreography model acts as a design-time artifact that defines multi-party interaction contracts (i.e. "to be" interaction processes) which can then be linked to existing (i.e. "as is") interaction processes provided by the individual parties and captured in their service interfaces (e.g. BPEL abstract processes linked to WSDL interfaces).

With the advancement of process-based service composition has come the need for consolidated insights into the capability and exploitation of the resulting standard specifications and associated implementations in terms of business requirements. The developments of BPEL and WS-CDL have been accompanied by requirements gathering and use cases; however these have largely steered towards technical concepts and implementation concerns, with documented use cases and examples, in the end, reflecting little more than simple processes involving basic "buyer-supplier-shipper" interactions. A major uncertainty faced by early and prospective adopters is whether the current proposals will go beyond simple paper-trails into large-scale production-level deployments.

This paper takes a small but important step towards dispelling this uncertainty. It sheds light into the nature of *service interactions* in collaborative business processes, where a number of parties, each with its own internal processes, need to interact with one another according to certain pre-agreed rules. The number of parties in such collaborative processes may be in the order of tens or even hundreds and thus the nature of interactions is rarely only bilateral (between two parties) but rather multilateral. Furthermore, the assumption of strict synchronization of all canvassed responses before the next steps in a process 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 (while further responses continue to be accepted). Also, while it is conventional to think of multicast interactions as a party sending a request to several other parties, the reverse may also apply: several parties send messages from autonomous events to a party which correlates these into a single request.

Another crucial feature is the fact that not all service providers in marketplaces have comparative advantage and collaborate. Not untypically, they offer similar services and therefore compete. The implication is that canvassed requests to competing service providers require exclusivity – e.g the first response that arrives is accepted while the others are not.

As a final observation, not all interactions in dynamic marketplaces follow a requestor-respondent-requestor structure. Rather, a sender may re-direct interactions to nominated delegates (or proxies). Receivers may outsource requests, choosing to "stay in the loop" and observe parts of responses. More generally, it may only be possibly to determine the order of interaction at run-time, given, say, the content of messages passed.

The paper takes a general approach to describing interactions reflecting these situations. Specifically, a set of *service interaction patterns* is proposed which apply primarily to the service composition layer but also to some extent to lower layers dealing with message handling and protocols. Patterns have proved invaluable in the reuse of requirements, design

and programming knowledge and expertise. They were traditionally the province of software design, e.g. the widely referenced object-oriented design patterns, but have more recently emerged in the BPM field, e.g. through the Workflow Patterns (van der Aalst et al. 2003). The value of patterns lies in their independence of specific languages or techniques. A pattern, as conventionally specified, captures the essence of a problem, collects references by way of synonyms, provides real examples of the problem, and proposes solutions for implementation in terms of concrete technologies. As such, patterns offer valuable problem-solving insights and allow a particular language's capability to be assessed. In particular, the proposed service interaction patterns have been used to analyze the scope and capabilities of BPEL and to some extent, of related specifications such as WSDL and WS-Addressing.

The collected patterns have been derived and extrapolated from insights into real-scale B2B transaction processing, choreography and orchestration examples including use cases gathered by standardization committees (e.g. BPEL and WS-CDL) during their requirements analysis phase, 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, but aims simply at consolidating recurrent scenarios and abstracting them in a way that provides reusable knowledge to service designers.

The proposed patterns are structured into four groups derived from 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*).
- 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*).

The first group of patterns 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 logical party. The final group is dedicated to routed interactions.

The proposed patterns may be composed through operators expressing control-flow dependencies such as sequence, choice, synchronization, etc. In this paper however, we do not deal with the composition of interaction patterns. Also, the focus is on capturing: (i) interactions between services and (ii) direct dependencies between interactions and internal actions (e.g. processing of an incoming message or preparation of an outgoing message). It is not in the scope of these interactions patterns to capture steps performed internally by a service that do not directly contribute to nor directly result from interactions. Also, we abstract away from data representation and manipulation issues, which deserve a separate elaboration as they encompass complex issues related to data mediation. Finally, the patterns do not cover security issues which again deserve a separate treatment.

Each pattern is made up of a description, a set of synonyms, examples, a discussion of the "forces" (i.e. issues and design choices) involved, a discussion of how the pattern can be approached using existing web services technology, and a discussion on related patterns and/or notions. The solutions are described at a high level of abstraction and do not include code fragments. A future version of this paper is expected to include detailed solutions.

The structure of the paper follows the groups of patterns outlined above.

2 Single-transmission bilateral interaction patterns

Pattern 1. Send

***Description*.** A party sends a message to another party.

Synonyms. Unicast; point-to-point send.

Example.

- An alerting service sends a reminder of an anniversary to a registered user.

Issues/design choices.

- The counter-party may or may not be known at design time. It may for example be selected at runtime from a list of possible parties, or retrieved from a local database, an LDAP registry or a UDDI directory.
- Reliable/guaranteed delivery may or may not be required. If reliable delivery is required, the send action may block until it is known that the message has been delivered, or it may be non-blocking. Also, in the case of reliable delivery the send action may result in a delivery fault.
- The message sent out may result in a fault message in response (not to be confused with a delivery fault as mentioned in the previous point).

Solution. Comprehensive solutions to this pattern are part of many messaging programming interfaces and platforms. For instance, the Message Passing Interface (MPI) (Snir & Gropp 1998) supports send actions with reliable delivery in blocking or non-blocking mode.

In the field of web services, a distinction is made between the communication-related aspects of this pattern and its control-related aspects. The latter are determined by the executable language used to implement the service (e.g. a general-purpose procedural language or BPEL). Reliability is determined at the communication level whereas synchronization (i.e. the blocking vs. non-blocking distinction) is determined at the level of the executable language.

In WSDL 2.0, the send pattern surfaces as two Message Exchange Patterns (MEPs): *out-only* and *robust out-only*. In the out-only MEP a service sends exactly one message. In the Robust out-only MEP the service also sends one message, but the message may result in a fault message in response. The fault message is delivered to the initiator of the interaction. For example, a purchase order sent to a supplier may result in a fault message from the supplier to the effect that the customer ID does not match the customer name.

Reliable delivery is the realm of a yet-to-be-standardized specification. Such reliable messaging specification is expected to provide the building blocks to enable *at-least once*, *at-most once*, and *exactly once* delivery. It will also define elements to specify the reliability properties of service operations in its interface (e.g. in a WSDL description). It should be noted though that the distinction between blocking and non-blocking interaction mode is not in the scope of WSDL, as this is determined at the level of the executable language.

In BPEL, the pattern is encoded either as an *invoke* activity that produces no output, or as a *reply* activity. A reply activity is always related to a previous receive activity and is used to respond to a previously received request. In an invoke/reply activity, the identity of the counter-party is captured by an "endpoint reference" defined for example using WS-Addressing.[4] There is an indirect relationship between endpoint references and invoke/reply activities: Each invoke/reply activity is associated with a *partner link* and each partner link is associated with an endpoint. The association between a partner link and an endpoint is determined at runtime. Thus the counter-party does not need to be known at design time.

In the case of a reliable delivery, the invoke/reply activity is blocking. Once it is certain that delivery has occurred, the invoke/reply activity completes normally and the flow of execution proceeds. Alternatively, a fault will be raised if the message fails to be delivered within a specified timeframe or after an inactivity period (these parameters may be defined as

[4] http://www.w3.org/2002/ws/addr

reliability policy assertions). To deal with delivery failures, the invoke/reply activity should be embedded in a so-called "scope" to which a suitable "fault handler" should be attached.

A BPEL invoke activity may be related to a WSDL operation[5] that may result in a fault message being returned by the counter-party. Again, to deal with such operation-defined faults, the invoke activity needs to be embedded in a scope with a corresponding fault handler. It can be noted that a reply activity cannot result in an operation-defined fault.

Pattern 2. Receive

Description. A party receives a message from another party.

Example.

- A purchasing service receives a notification of delivery delay from a shipping service.

Issues/design choices.

- The sender may require the receiver to support a reliable delivery protocol (e.g. provide receipt acknowledgments and discard duplicates).
- An incoming message may be received at a moment when the intended recipient is not ready to consume it. In this case, the message may be discarded (and a fault message sent back to the sender) or it may be placed in a queue with the expectation that it may be consumed in the future. If queuing is chosen, choices need to be made regarding queue management. For example: How long should a message be kept in a queue? Should there be a garbage-collection mechanism to clean the queue regularly, and if so, what criteria are used to discard messages?

Solution. As for the send pattern, comprehensive built-in solutions to this receive pattern can be found in many messaging programming interfaces and platforms (e.g. MPI). Also as discussed in the previous pattern, web services standards and platforms separate the communication aspects of this pattern from the control aspects.

On the communication side, WSDL supports the pattern through two MEPs: in-only and robust in-only, depending on whether a fault message may be sent back to the sender or not. The receiver's capabilities in terms of reliable delivery (e.g. being able to ensure that duplicate messages are only processed once) are exposed as reliability policy assertions in a yet-to-be-standardized specification. Due to its stateless nature, WSDL cannot express under which conditions the message is consumed or discarded (e.g. it does not capture which message exchanges must have occurred before a given type of message can be consumed). The formulation of such preconditions is pushed to choreography and behavioral interface descriptions languages (e.g. WS-CDL and BPEL abstract processes). However, at the WSDL level it is possible to define fault messages that will be returned to the sender if a given message is discarded, for example because it arrives too late or too early in an interaction.

From the control perspective, the receive pattern can be treated either as an event that is produced when the message is ready for consumption, or as an action that checks if message has arrived, waits for the message if necessary (i.e. the execution is blocked), and eventually consumes the message. Accordingly, the receive pattern can be captured in BPEL either as an *onEvent handler* or as a *receive* activity depending on whether the message receipt is non-blocking or blocking respectively. A *receive* activity blocks until the message is consumed. On the other hand, an *onEvent* handler is associated to a "scope" of the process and may be triggered anytime during the execution of the scope by a *message event* matching the handler, thus capturing a non-blocking receive. It can be noted that, in addition to the onEvent handler, BPEL provides another form of message event handler, namely *onMessage handler*. Handlers of this latter form are used as triggers within "pick" activities when an exclusive choice between multiple events needs to occur as in the "Racing messages" pattern exposed later.

[5] The current version of BPEL is linked to WSDL 1.1 rather than WSDL 2.0, but this is not relevant to this discussion.

Central to BPEL is the notion of *process instance*, i.e. an execution of a process definition that runs in its own memory space. An incoming message to be handled by a BPEL process may either trigger the creation of a new process instance, or it may be routed to an already running process instance based on the value of a *correlation token* contained in the message. Whether a message creates a new instance or is handled by an existing instance depends on the value of an attribute (namely *createInstance*) attached to the message event handler or receive action that is able to receive the type of message in question. Specifically, when a receive activity or message event handler has the createInstance attribute set to "yes", a message matching this handler/activity will not be consumed by a process instance, but rather by the process factory that will create a new process instance as a result.

When a message is targeted at a running process instance, if the process instance is not ready to consume the message upon arrival (i.e. there is no *active* receive action or message event handler capable of consuming that type of message), the message is stored until the process instance reaches a point where it can consume it. In theory, the message could be stored forever if the process instance completes without reaching such state. Thus a typical BPEL implementation would provide some "garbage collection" mechanism and discard messages that would otherwise never be consumed. BPEL regards this as an implementation issue. In particular, when messages are discarded in this way, BPEL does not stipulate that a "fault message" should be returned to the original sender to indicate that its message has been discarded.

Pattern 3. Send/receive

Description. A party X engages in two causally related interactions: in the first interaction X sends a message to another party Y (the request), while in the second one X receives a message from Y (the response).

Synonyms: Request/response**.**

Example.

- A payment service sends a payment to a retail service provider who either sends back a message indicating that the payment details are invalid, or a receipt.

Issues/design choices.

- The counter-party may or may not be known in advance.
- The outgoing and incoming messages must be correlated. In other words, there is a common item of information in the request and the response that allows these two messages to be unequivocally related to one another.
- Either of the two interactions may result in a fault message in response.
- The sender may block a thread of execution to wait for the response or fault, or it may provide a single continuation for both the response and the fault or two separate continuations.

Solution.

WSDL provides two MEPs corresponding to this pattern: out-in and out-optional in. The former corresponds directly to this pattern, while the latter corresponds to a combination of this pattern and the send pattern.

In BPEL, the pattern can be captured in two ways: (i) as a single invoke activity with both an input and an output, in which case the thread of execution is blocked until the response arrives; or (ii) as a combination of an invoke activity (without an output) and either a receive activity or an onEvent or onMessage handler referring to the same partner link, operation, and correlation set as the invoke activity. Through this latter option, it is possible to deal with the pattern in a non-blocking mode, in the sense that the thread of execution would not block between the send and the receive actions, and in addition (by using onEvent handlers) the message receipt itself is non-blocking. Also, if the send and receive parts of the pattern are realized separately, it is possible to determine the endpoint to which the initial message is sent at runtime (as explained in the solution of the Send pattern) and thus the counter-party does

not need to be known at design time. Faults can be taken into account through scopes with associated fault handlers as mentioned in the Send and Receive patterns above.

Related pattern:

- Remote Procedure Call (RPC) [Hohpe & Woolf 2003]. RPC refers to the ability for a process to invoke a procedure (or function) that will be executed (or evaluated) in another possibly remote process. When the remote procedure is invoked, parameter values are passed to it. The corresponding thread of execution blocks on the caller's process until the function/procedure has been executed/evaluated, and then the outputs are made available.
- Receive/send: This is the dual of the send/receive pattern: A party X engages in two causally related interactions: in the first interaction X receives a message from another party Y, while in the second one X sends a message to Y. For example, a supplier receives a request for quote and replies with a quote or a fault indicated that the request for quote is invalid. The issues and design choices associated to the receive/send pattern are analogue to those of the send/receive. The same holds for the solution: WSDL provides two MEPs corresponding to the receive/send pattern (in-out and in-optional out), while in BPEL this pattern is handled as a pair receive/reply action referring to the same partner link, operation, and correlation set.

3 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.

Synonyms. Racing messages (Kilgore & Chase 1997).

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.
- The escalation service of an insurance company's call center may receive storm alerts from a weather monitoring service (which typically herald surges in demand), notifications of long waiting times from the queue management service, or notifications of low resourcing levels from the call center's HR manager. The receipt of any of these three types of messages by the escalation service triggers an escalation process (different processes apply to the various types of notifications). While an escalation process is running, subsequent storm alerts, queue saturation or low resourcing notifications are made available to the call center manager but will not trigger new escalations.

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 then immediately consumed by the handler. The pick enforces that at most one of its associated onMessage handlers will consume an event. Note that it is also possible to associated 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 (van der Aalst et al. 2003). 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 "competing receives" 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 (Snir & Gropp 1998).

Examples.

- A purchasing service sends a call for tender to all known trading parties that provide a given type of product or service.
- An accreditation authority sends a notification to all registered candidates who have passed a certification test. Each of these notifications contains information that is specific to the recipient (e.g. test results).

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 described above 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:

- In BPEL the individual send actions would have to be scheduled sequentially. This can be achieved through a "while" loop in which one message is sent at each iteration, with an associated fault handler if necessary. The sequential nature of this solution is problematic when the individual send actions are blocking, since the second and subsequent send actions need to wait for all the preceding ones to complete before starting. Note that if the number of parties is known at design time, it is possible to capture this pattern 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.
- In certain proprietary extensions of BPEL, such as Oracle BPEL[6], a "parallel while" construct is provided that allows to execute the same activity an arbitrary number of times in parallel, such that this number is only determined when the "parallel while" block is reached (this construct is called "FlowN" in Oracle BPEL).[7] The pattern can be directly captured by a "parallel while" block in which the activity inside the "parallel while" is a send action and its associated fault handler if necessary.
- In WSCI and BPML[8], 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.

This pattern requires a mechanism supporting "dynamic binding by reference" (Alonso et al. 2003) 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 "endpoints references" (i.e. descriptors of service port locations) as first-class citizens that can be associated with predefined partner links at runtime. These partner links are then associated with send actions in the process definition. Endpoint references can be described in WS-Addressing.

Related patterns:

- *Broadcast*. While the terms broadcast and multicast are sometimes used interchangeably, the term broadcast is most appropriately used where a message is sent to an open set of parties, i.e. the sender does not know who the recipients will be. Prospective recipients must either subscribe with a broker to receive that type of message (as in publish/subscribe messaging middleware) or access the message from a message board or equivalent mechanism. Typically, a broadcast is achieved by sending the message to a broker or message board service, which then pushes it or makes it available to the final recipients. Thus, a broadcast can be expressed as a one-to-one send from the sender to the broker, possibly followed by a one-to-many send from the broker to the final recipients.

---

[6] Downloadable from: http://www.oracle.com/technology/products/ias/bpel. This paper refers to version 2.1.2 of the Oracle Process Manager and version 0.9.10 of the process designer.

[7] A similar construct (namely "bundle") has been proposed for introduction into the BPEL standard, but this proposal has not been adopted; see Issue 4 in the list of issues available from OASIS BPEL TC web page (http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel).

[8] http://www.bpmi.org

- *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 a number of logically related messages that arise from autonomous events occurring at different parties. The arrival of messages needs to be *timely* enough for their correlation as a single logical request. The interaction may complete successfully or not depending on the set of messages gathered.

Synonyms: Event aggregation (Luckham 2002), gather (Snir & Gropp 1998).

***Examples*:**

- *Multi-part order*. A printer operating in a high-demand market allows a streamlining of job preparation whereby documents belonging to the same job can be supplied directly by different parties (as with legislative documents by governments requiring quick turn-around). The first notification carries the job requirements including the delivery time and location. This and subsequent messages provide document content. Several interactions take place between the printer and the primary partner to mark-off the production steps, e.g. typesetting preview and payment. The printer's policy has a deadline for collection of all source documents in order to meet production deadlines. If the sources are not available by this deadline, the interaction fails, otherwise it succeeds. (More complex variant of the purchase order example given in (Luckam 2002) page 78).
- *Batched requests*. 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.
- *Event filtering prior to persistence*. Investment consultants subscribe to a stock market watch service which broadcasts significant trading events. This allows the consultants to provide timely recommendations to their customers for changes in share portfolios. Events are correlated into composite events per fund manager. Because investment consultants are typically small enterprises (or smaller units of large enterprises), they cannot afford large databases in order to persist the high-volumes of incoming events prior to correlation. Rather they apply correlation rules as the events become available. If the individual events successfully correlate within the relevant timeframe into a significant composite event, an escalation is triggered. Otherwise the events are ignored.

***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, in one way or another, a notion of *stop condition* in the solution to this pattern. In the general case, 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 (and in particular, when the first message of a group arrives). 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 and $|\ldots|$ denotes the cardinality of a set.

To be complete, a solution to the pattern should associate timers to message groups. The timer for a group is started when the group is created. In many usage scenarios of this pattern the group is created when the first message mapping to the corresponding group ID is received. To simulate the case where a group is explicitly created by the receiving service, the service may send a "dummy" message to itself with the corresponding group ID, thus forcing the creation of the group. This "dummy" message will have to be removed or abstracted away after the group is completely formed and when evaluating the stop and success conditions. In any case, when the timer expires, the group is considered to be complete and the corresponding group ID may be flagged as no longer valid. Note that it is possible that a timeout occurs even if no message (except for the "dummy" one) 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" or "product" of the interaction.

To further illustrate the solution, we consider the examples introduced above:

- For "multi-order", the "stop condition" is a conjunction of all document source types needed (each of which can be correlated against expected incoming document sources), the "success condition" is that all expected document sources should have arrived by the printer's deadline, a group is created when the first message for that group is received.
- For "batched requests", the "stop condition" and the "success predicate" 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 correlation IDs are never flagged as invalid since it is always possible to process requests for a given type of product whether or not previous groups for the same type of product have been filled or not.

- For "event filtering prior to persistence", the "success predicate" and the "stop predicate" both have the composite event condition.

Related pattern:

- *Multiple instances with a priori runtime knowledge (MIRT)*. See discussion in the "Related patterns" paragraph of the previous pattern. Note that existing 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 (i.e. the number of messages received).

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 (Snir & Gropp 1998, Hohpe & Woolf 2003).

***Example*:**

- *RosettaNet Partner Interface Process (PIP) 3A3 – "Request Price and Availability"* (http://www.rosettanet.org/PIP3A2). In this process, a buyer identifies a number of potential suppliers and sends a request for "price and availability" to each of them. Responses from all of these suppliers are then gathered and analysed. The number of potential suppliers is not known at design time.
- 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[9], 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.

[9] Specifically, in BPEL the invoke/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.

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 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 one-to-one send/receive through parallel composition, such that the resulting composition may be interrupted 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, specific fault handling routines (i.e. fault handlers in BPEL) 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 successfully delivered, so that this information can be taken into account in the stop and success conditions. In this way, it is possible to express success conditions such as "stop as soon as half of the parties who actually received a request have responded".

Related pattern:

- Scatter-gather (Hohpe & Woolf 2003). The scatter-gather pattern is a special case of the one-to-many send/receive. The scatter-gather assumes that all parties will respond in a timely manner and that all responses must be gathered and thus it does not address issues related to timeout, stop conditions, and success conditions.
- One-from-many receive/send. This is the dual of the one-to-many send/receive: A party receives messages from a number of other parties, correlates and processes them collectively, and constructs responses for all the parties. For example, a tendering service collects bids from various parties for a given period of time. At the end of this period, a winner is determined and all the parties who placed a bid are notified of the outcome. The issues and design choices are analogue to those of the one-to-many send/receive. The same holds for the solution which can be obtained by combining the "one-to-many receive" pattern with the "one-from-many send" through sequential composition.

4 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[10]; (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*:**

[10] An obvious example in email is the "unsubscribe" contained in a message body by a list subscriber to terminate the subscription. Messages sent to that list are not sent, thereafter, to that address.

- *Order forecasting.* As part of order forecasting, a buyer sends a request to a seller providing point-of-sale data, events impacting on order forecast, e.g. new products, stores opening/close, inventory strategy data and current inventory position (on hand, in transit, on order). The seller provides response data as various investigations make that data available. The investigations relate to subcontractors in manufacturing and delivery, market studies (historical demand data), and logistical delivery data (capacity related, lead times, delivery operations schedules).
- *News refresh.* 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 the Racing Messages pattern, the messages to be received may be of different types. The way each received 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 with the Racing messages pattern, the core of this pattern can be captured in BPEL through a pick activity with one onMessage handler per type of message expected (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 as follows:

- 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 respondent (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 the *while/scope* activity mentioned above, which upon receiving any of the expected types of

messages from Y, will return 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. The BPEL specification is silent as to what should be done with messages that are received by a process but not consumed by any of its activities.

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 alternatively sends a request to another party Z, and so on.

Synonyms. Send with failovers.

Examples.

- An online service supports document submissions to academic conferences, tenders and grant/funding schemes. Applicants span different geographic boundaries for the different applications. Large bottlenecks are experienced near deadlines of especially high-volume applications. Proposals can be directly uploaded wherein further interactions are required to entry and validation of proposal details. Alternatively, different servers are available in different geographic localities for queue/forward of proposals to the central repository. The validation steps of proposals take place once the proposal has been forwarded (a notification is sent to the applicant to trigger this). In first instance, the system attempts to submit the proposal for direct upload to the central repository, but if this service is unavailable or overloaded, it will try the "nearest: queuing service, and so on with other queuing services.
- 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. It is concerned with contingent sending only. 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 atomicity.* A business venture service[11] supports the process of business license applications for various small business endeavors (e.g. 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 (down to the same location). 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.
- *Competing through atomicity.* 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 (service level agreements, e.g. cots model and timeliness of response obligations). 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.
- *Nesting.* A travel agent allows the booking of both international and domestic travel requirements as part of comprehensive travel packaging. Customers nominate their preferred flight carriers as well as domestic requirements such as hotel accommodation, local travel bookings/hire and reservations to key attractions. In this example, each node in the international flight path can be seen as atomic group, and within a group, the flight carrier, booking agencies for local hotels, travel, care hire and so on, identify the services contacted. The different groups can have different minimum and maximum constraints depending on the domestic requirements of the customer. However, all atomic groups need to succeed in order for the interaction as a whole to succeed. (It should be noted that an extension of this example could see further levels of nesting within the atomic groups).

Issues/design choices:

- The exact number of parties to which the notification will be sent may be known at design time or at run-time.
- Specification for the minimum and maximum constraints 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.

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 (Hagen & Alonso 2000) 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

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

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[12], which provides a realization of the distributed two-phase commit (2PC) protocol. However, this specification has not yet matured into a standardization initiative and accordingly no links exist between it and BPEL.

Arguably, a solution could be crafted in BPEL by encoding the 2PC protocol as a sequence of sub-interactions. For the first phase, a PREPARE control message is sent to each party. Each receiver has a separate process to deal with this message which eventually will send back a READY control message to the initiator of the transaction. The responses are tallied after the timeout to determine whether the minimum and maximum constraints are satisfied. After this, the second phase has a related set of processes for each party providing a COMMIT (go-ahead and process the notification) or REJECT message. It should be noted that different payloads may be included in the first and the second phases. As part of the first phase of interactions, contacted parties might only see a limited content of the message, enough to decide whether they are ready to *accept* the request or not. In the second phase, selected parties see all details needed to *act* on the request.

If the maximum number of parties that are required to commit is less than the total number of parties to which the request is sent, it is necessary to introduce a "preference function". Where more parties than the maximum allowed express interest to accepting the notification, the requestor would select a subset of them, corresponding the maximum number allowed, using this preference function.

5 Routing patterns

Pattern 11. Request with referral

***Description*:** Party A sends a request to party B indicating that any follow-up response should be sent to a number of other parties (P1, P2, …, Pn) depending on the evaluation of certain conditions. While faults are sent by default to these parties, they could alternatively be sent to another nominated party (which may be 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/activities at these parties.

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

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 between the parties must allow the referred parties to route the message to the correct internal process/activity.

Solution:

At the messaging level, this pattern is addressed in part by WS-Addressing[13] 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* (Fielding 2000): (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 (e.g. the activity that expects shipment status notifications for that purchase order), while in the second style, this data item would be encoded inside the shipment status notification.

At the service composition level (specifically in BPEL), endpoint references can be manipulated as ordinary data. In particular they can be included in the contents of a message. They can also 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 activities. 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 (Milner 1999) refers to the ability for a process X to pass a channel name to another process Y. Passing channel names along with requests, and associating this channel name with "action identifiers" (like the "reply-to" field in WS-Addressing) 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:

[13] http://www.w3.org/2002/ws/addr

- The delegated parties (P1, …, Pn) may or may not have prior knowledge of the identity of the request originator, party A. The information transferred from party B to the delegated parties must therefore allow these to fully identify and interact with 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 exchanges.
- Party B could apply referrals for monitoring interactions or faults to other parties, however this issue is orthogonal to this pattern (and has been covered in request with referral, pattern 11).
- The view will be defined during design time, but could 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 monitoring interactions or faults to other parties, however this issue is orthogonal to this pattern (and has been covered in request with referral, 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-Addressing 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.

The WS-Addressing strategy conjures up the possibility of including party B in interactions through a "Cc" field. Apart from the fact that WS-Addressing currently does not support a "Cc" field, this suggestion compromises an important 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 what they are obliged to reveal to B, as required by the pattern. A corollary of this requirement is that some message content between the parties remains oblivious to B.

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". Of course, these need to be coordinated with A, so that both ends of interactions are subject to the new "view" version. 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 would need to 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[14] 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

[14] At a minimum, routing conditions should allow late-binding to concrete service endpoints to be supported. It is envisaged that interleaved parallel routing, as described in the Workflow Patterns, be supported.

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 (Hohpe & Woolf 2003, Kumar & Zhao 2002), Content-based router (Chaterjee 2004).

***Example*:**

- *Flexible order fulfillment*. 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.
- *Proposal reviews.* A project proposal initiated by a project coordinator is required to be passed through its work-package coordinators in any order, one at a time. For each route, all coordinators get copies of the document, however only one, i.e. the first expressing interest, is allowed to do the update. A coordinator updates the document and makes it available for the next coordinators' "read only" copy, out of which one gets "write" access. After an update, a problem may be flagged which requires the proposal to be routed back to the project coordinator. This over-rides the next step of the routing, and a modification of the routing may be issued by the project coordinator.
- *Legal case preparation.* A legal case preparation service is utilized by law courts to reduce the number of hearing re-schedules. This is a costly problem for the courts and defers justice for litigants due to insufficient information to embark on hearings, decreed by judges typically within the first moments of a hearing. As part of improved preparation, the clerk of the court examines a scheduled hearing, obtains all relevant documents into a formal draft, and determines the relevant legal or administrative actors required to provide examine the draft for verification and additional input of the draft. The clerk determines the first set of actors (defense and prosecution lawyers, and courts jurisdictionally related to the case) to review the draft. After these, expert opinion is canvassed based on issues raised in the investigation, e.g. different departmental solicitors in different categories of expertise.

***Issues/design choices*:**

- The set of parties through which the request should circulate might not be known in advance. Moreover, these parties may not know each other at design/build time.
- The specification of ordering should support service/role late binding flexibility, parallelism and inter-leaved parallel routing, synchronization points between parallel steps, and dynamic conditions.
- A way of providing relevant fragments of documents to different parties needs to be supported.
- A way of controlling read-only and write access of documents provided to parties needs to be supported as does a way for readers to get the "write" token.
- 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. Hand-crafted BPEL solutions are possible by allocating structured fields for deriving routing conditions. Routing could then be expressed as a (central) BPEL workflow and events could be defined to be matched against an incoming document from the current step of routing.

Permutations of events will have to be coded for the different roles at different steps and the state of the routing thus far. The next roles in routing can be assessed through events, and activities are triggered for these. Of course, this is an convenient solution, to say the least, due to sheer permutation of events by different in different routing states. Furthermore, application coding for document synchronization needs to be provided, noting that this, while required by dynamic routing, is orthogonal to it.

WS-Routing[15] (which is 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 "routing slip". However, the complex dynamic routes required by our examples above, cannot currently be supported.

6 Conclusion

As business process management and service composition developments unfold in their objectives of both 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 stimulating and convincing insights in business requirements. Use cases have been gathered through the relevant standardization groups (like BPEL, RosettaNet and WS-CDL), however these are far from comprehensive and weigh either on technical aspects or business contexts which reveal little in terms of possible extensions to languages' expressive power.

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 and standardization activities, and extrapolating from these. The result is a set of patterns which on the one hand consolidates the nature of service interactions through generalized functional classification, and on the other clears the track for further and ongoing extensions. It is our hope that such a reference can be of similar value pro-offered by previous patterns collection efforts - notably the body of patterns available in object-oriented design and workflow management. Importantly, the patterns allow relevant technologies to be benchmarked for their ability to capture the patterns. In this paper, we have investigated BPEL's capabilities.

BPEL directly supports *single-transmission bilateral patterns* (not presented in this paper). 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" 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, the lack of sufficient transaction support significantly compromises a BPEL solution for *atomic multi-cast*.

For the *routing patterns*, simple *request referrals* are possible through by passing endpoint references of delegates/proxies, and securing integrity of indirect interactions through correlation identifiers. This also serves *request relaying*, and message filtering can be implemented through XPATH/XQUERY. As "view" conditions take on a more dynamic nature, burden of application coding becomes larger. Also we observed, WS-Addressing could used for request referrals although it cannot support request relaying due to the current specification not having a "Cc". *Dynamic routing* is outside the scope of BPEL and WS-Routing can serve to implement some aspects of it. This is probably the most complex pattern due to the intricate parallel/synchronization pathways and routing override through dynamic conditions, going beyond WS-Routing and other service composition technologies.

Future work will extend the patterns by further extrapolations (many-to-many send/receive) and will consider conversation management, viz. create conversation, stop conversation,

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

conversation suspension and resumption, cancel and undo conversation and compensate conversation. We are also drawing on insights from the patterns for conceptual modeling of service interactions. A conceptual level, as such can different technologies to be leveraged as they become available.

Acknowledgments. The authors wish to thank Phillipa Oaks and Helen Paik for their contributions to the initial phases of this work, as well as Ivana Trickovic for her highly valuable feedback. The work of the second author is funded by a Queensland Government "Smart State" Fellowship co-sponsored by SAP.

References

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., & Barros, A. (2003). Workflow Patterns. *Distributed and Parallel Databases*, *14*(1):5-51. See also http://www.workflowpatterns.com.

Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2003). *Web services: Concepts, architectures and applications*. Springer Verlag.

Chaterjee, S. Messaging Patterns in Service-Oriented Architecture (Parts 1 and 2). *Microsoft Architects Journal, Issues 2 and 3*, April and July 2004. Available from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmaj/html/messagingsoa.asp and http://msdn.microsoft.com/library/en-us/dnmaj/html/aj2mpsoarch.asp

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

Hagen, C. & Alonso, G. (2000). Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering* 26(10): 943-958.

Hohpe, G. & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.

Kilgore, R. & Chase, C. (1997). Testing Distributed Programs Containing Racing Messages. *Computer Journal* 40(8): 489-498.

Kumar, A., and Zhao, L. (2002). Workflow Support for Electronic Commerce Applications. *Decision Support Systems* 32: 265-278.

Luckham, D. (2002). *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley.

Milner, R. (1999): *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press.

Snir, M. & Gropp, W. (1998). *MPI: The Complete Reference*. MIT Press, 2nd edition.