COVER SHEET

Correlation Patterns in Service-Oriented Architectures

Alistair Barros[1], Gero Decker[2⋆], Marlon Dumas[3], Franz Weber[4]

[1] SAP Research Centre, Brisbane, Australia
alistair.barros@sap.com
[2] Hasso-Plattner Institute, University of Potsdam, Germany
gero.decker@hpi.uni-potsdam.de
[3] Queensland University of Technology, Brisbane, Australia
m.dumas@qut.edu.au
[4] SAP AG, Walldorf, Germany
franz.weber@sap.com

Abstract. When a service engages in multiple interactions concurrently, it is generally required to correlate incoming messages with messages previously sent or received. Features to deal with this correlation requirement have been incorporated into standards and tools for service implementation, but the supported sets of features are ad hoc as there is a lack of an overarching framework with respect to which their expressiveness can be evaluated. This paper introduces a set of patterns that provide a basis for evaluating languages and protocols for service implementation in terms of their support for correlation. The proposed correlation patterns are grounded in a formal model that views correlation mechanisms as means of grouping atomic message events into conversations and processes. The paper also provides an evaluation of relevant standards in terms of the patterns, specifically WS-Addressing and BPEL, and discusses how these standards have and could continue to evolve to address a wider set of correlation scenarios.

1 Introduction

Contemporary distributed system architectures, in particular service-oriented architectures, rely on the notion of message exchange as a basic communication primitive. A message exchange is an interaction between two actors (e.g. services) composed of two events: a message send event occurring at one actor and a message receive event at another actor. These events are generally typed in order to capture their purpose and the structure of the data they convey. Example of event types are "Purchase Order", "Purchase Order Response", "Cancel Order Request", etc. Event types are described within *structural interfaces* using an interface definition language such as WSDL [1]. Sometimes, message exchanges are related to one another in simple ways. For example, a message exchange corresponding to a request may be related to the message

⋆ Work conducted while the author was visiting SAP Research Centre, Brisbane

exchange corresponding to the response to this request. Such simple relations between message exchanges are described in the structural interface as well (e.g. as a WSDL operation definition).

The above abstractions are sufficient to describe simple interactions such as a weather information service that provides an operation to request the forecasted temperature for a given location and date. However, they are insufficient to describe interactions between services that engage in long-running business transactions such as those that arise in supply chain management, procurement or logistics. In these contexts, message event types can be related in complex manners. For example, following the receipt of a purchase order containing several line items, an order management service may issue a number of stock availability requests to multiple warehouses, and by gathering the responses from the warehouses (up to a timeout event), produce one or several responses for the customer. Such services are referred to as *conversational services* as they engage in multiple interrelated message exchanges for the purpose of fulfilling a goal. Conversational services are often related to (business) process execution, although as we will see later, conversations and processes are orthogonal concepts.

The need to support the description, implementation and execution of conversational services is widely acknowledged. For example, enhancements to the standard SOAP messaging format and protocol [1] for correlating messages have been proposed in the context of WS-Addressing [3]. WS-Addressing is now supported by the APIs of most service-oriented middleware. However, WS-Addressing merely allows a service to declare (at runtime) that a given message is a reply to a previous message referred to by an identifier. This is only one specific type of relation between interactions that has a manifestation only at runtime (i.e. it does not operate at the level of event types) and fails to capture more complicated scenarios where two message send (or receive) events are related not because one is a reply to another (or is caused by another), but because there is a common event that causes both. This is the case in the above example where the stock availability requests are related because they are caused by the same purchase order receive event.

Another upcoming standard, namely WS-BPEL [2], provides further support for developing conversational services. In particular WS-BPEL supports the notion of *process instance*: a set of related message send and receive events (among other kinds of events). Events in WS-BPEL are grouped into process instances through a mechanism known as *instance routing*, whereby a receive event that does not start a new process instance is routed to an existing process instance based on a common property between this event and a previously recorded send or receive event. This property may be the fact that both messages are exchanged in the context of the same HTTP connection, or based on a common identifier found in the WS-Addressing headers of both events, or a common element or combination of elements in the message body of both events. Thus, WS-BPEL allows developers to express event types, which are related to WSDL operations, and to relate events of these types to process instances. It also allows developers

to capture ordering constraints between events related to a process instance, which ultimately correspond to causal dependencies (or causal independence).

Despite this limited support for message event correlation, there is currently no overarching framework capturing the kinds of event correlation that service-oriented architectures should support. As a result, different approaches to event correlation are being incorporated into standards and products in the field, and there is no clear picture of the event correlation requirements that these standards and products should fulfill.

In this setting, this paper makes three complementary contributions:

- A unified conceptualization of the notions of conversation, process and correlation in terms of atomic message events (Section 2).
- A set of formally defined *correlation patterns* that cover a spectrum of correlation scenarios that occur in the context of conversational services (Sections 3, 4 and 5).
- An evaluation of the support for these correlation patterns offered by relevant Web service standards, namely WS-Addressing and BPEL versions 1.1 and 2.0 (Section 6).

Together, these contributions provide a foundation to guide the design of languages and protocols for conversational services.

2 Classification Framework

When talking about correlation we mainly deal with three different concepts: events, conversations and process instances. An event is an object that is record of an activity in a system [4]. Events have attributes which describe the corresponding activity such as the time period, the performer or the location of the activity. We assume that a type is assigned to each event. In the area of service-oriented computing, where emphasis is placed on communication in a distributed environment, the most important kinds of events include message send and receipt events and time-related events (time-outs). In addition to these "communication events" that allow to observe the public behavior of actors, we deal with "action events" being records of internal activities within actors. Message send events are results of internal actions and most message receipt events result in internal actions *consuming* these events. Therefore, we assume that event logs include information about the causal relationships between communication events and action events. The causal relationship between corresponding message send and receipt events is also used in the remainder of this paper. Figure 1 illustrates this.

Events can be grouped in different ways, e.g. all events occurring at one particular actor could be grouped together. Since this work deals with event correlation in the context of conversational services, we focus on two types of event grouping: conversations and process instances. Conversations are groups of communication events occurring at different actors that all correspond to achieving a certain goal. Boundaries of conversations might be defined through

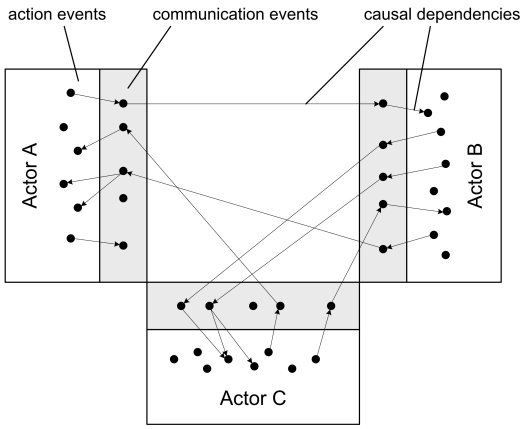

Fig. 1. Action and communication events

interaction models (choreographies) or might not be defined in advance but rather discovered a posteriori. Process instances are groups of action events occurring at one actor. Boundaries of process instances are determined by process models.

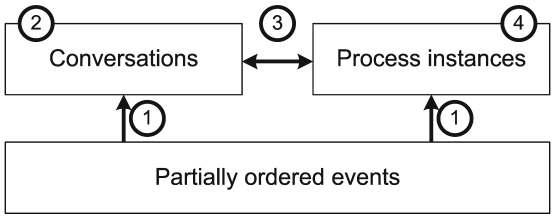

Fig. 2. Framework for classifying correlation patterns

Figure 2 illustrates a framework for classifying correlation patterns. At the bottom there are partially ordered events. The partial order stems from the temporal ordering of events occurring within one actor, combined with the relationships between a send event occurring at one actor and the corresponding receive event occurring at another actor. Since events may be recorded by different actors having clocks that are not synchronized we might not be able to linearly order all the events according to their timestamps. However, we can use the timestamps to order events that were recorded by the same actor (assuming a perfect clock within one actor). In the case of different clocks within one actor due to decomposition into components, causal relationships between action events occurring in different components can replace the pure timestamp-based ordering. For establishing a partial order between events that were recorded by

different actors we then use the relationships between corresponding message send and receipt events.

Conversations and process instances are sets of correlated events. The different patterns describing the relationships between events, conversations and process instances are grouped into four categories (for numbering see Figure 2).

1. Mechanisms to group events into conversations and process instances. These *correlation mechanisms* will be presented in section 3.
2. After having identified conversations we can examine how conversations are structured. In previous work we have investigated common interaction scenarios between participants within one conversation (cf. the Service Interaction Patterns [5]). The *conversation patterns* in section 4 present relationships between different conversations.
3. Relationships between conversations and process instances are covered in the *process instance to conversation mappings* in section 5.
4. Common structures within one process instance have already been extensively studied e.g. in the workflow patterns ([6]). Additional work has to be done to identify patterns describing the relationships between different process instances, but this is outside the scope of this paper.

Below, we present a set of formally defined correlation patterns. The formal description is based on the idea of viewing events from a post-mortem perspective. This could be seen as analyzing logs of past events. This view is taken for the sake of providing a unified formal description. In practice the patterns will not be necessarily be used to analyze event logs, but rather to assess the capabilities of existing languages that deal in one way or another with correlation in SOAs. A language will be said to support a pattern if there is a construct in the language (or a combination of constructs) that allows developers to describe or implement services which, if executed an arbitrary number of times, would generate event logs that satisfy the conditions captured in the formalization of the correlation pattern.

In the rest of the paper, we use the following formal notations:

- E is the set of events
- $CE, AE \subseteq E$ are the communication and action events ($CE \cap AE = \emptyset$)
- A is the set of actors
- function $does : E \rightarrow A$ links an event to the actor who records the corresponding activity
- $<_t \subseteq E \times E$ partially orders the events occurring at the same actor according to their timestamps
- $<_c \subseteq E \times E$ is the causal relation between events, including pairs of corresponding send and receipt events as well as corresponding communication and action events where $\forall e_1, e_2 \in <_c \ [does(e_1) = does(e_2) \rightarrow e_1 <_t e_2]$
- $<$ is a partial order relation on E being the transitive closure of $<_t$ and $<_c$: $<:= (<_t \cup <_c)^+$.
- $Conv \subseteq \wp(CE)$ and $PI \subseteq \wp(AE)$ are sets of sets of communication and action events corresponding to groupings of events into conversations (Conv)

and process instances (PI), respectively. These sets will in principle be generated using correlation mechanisms as discussed below.

3 Correlation Mechanisms

The correlation mechanism patterns focus on how events can be correlated to different process instances and more importantly to different conversations.

The purpose of correlation is to group messages into traces based on their contents (including message headers). Current web service standards do not make a provision for messages to include a "service instance identifier", so assuming the existence of such identifier may be unrealistic in some situations. Other monitoring approaches in the field of web services have recognized this problem and have addressed it in different ways, but they usually end up relying on very specific and sometimes proprietary approaches. For example the Web Services Navigator [11] uses IBM's Data Collector to log both the contents and context of SOAP messages. But to capture enable correlation, the Data Collector inserts a proprietary SOAP header element into messages. In the literature on correlation, it is often noted that correlation is different from causation. Correlation in general means that an event is (perceived to be) related to another in some way, while causation means that an event is a cause of another, which is a special type of relation. Thus, while causation implies correlation, the reverse does not hold. It is not in the scope of this paper to capture a general notion of correlation. The focus of the paper is on correlation between events in service-oriented architectures. In this context, we postulate that two events can be correlated in either of the following situations:

1. One event is a cause of the other, either directly or transitively.
2. There is a third event which is a cause (either directly or transitively) of both events.
3. Both events are a common cause (either directly or transitively) of a third event.
4. Both events have a common property, e.g. there exists a function that when applied to both events yields the same value. For example, two events can be correlated simply because they are performed by the same actor, or because they refer to the same purchase order.

In order to capture all these four types of correlation, we introduce two notions: *key-based correlation* (also called *function-based correlation*) and *chained correlation*. Different flavors of both mechanisms will be presented. The application of a particular key-based or chained correlation mechanism or a combination of different mechanisms leads to a *correlation scheme*. Such schemes are sets of sets of correlated events that might be interpreted e.g. as conversations or process instances later on. Different combinations are discussed in this section.

3.1 Function-based Correlation

Functions assign labels to an event. Events with common labels are then grouped together. We distinguish:

- ***C1. Key-based correlation***. One or a set of unique identifiers are assigned to an event and all events having at least one common identifier are grouped together. E.g. a process instance identifier and conversation identifier is attached to an event. Identifiers can be single values or compositions of several values. WS-CDL introduces the notion of identity tokens for channel instances that can be used for distinguishing different conversations. In BPEL we can find correlation sets being implemented as special fields in a WSDL specification. These correlation sets are an example for composite identifiers.
- ***C2. Property-based correlation***. A function assigns a label to an event depending on the value of its attributes. In contrast to key-based correlation not only equality can be used in the function. Operators such as "greater", "less", "or" and "not" must be available in the function. E.g. all events involving customers living less than 50km away from the city centers of Brisbane, Sydney or Melbourne are grouped together (label = "metropolitan") as opposed to the others (label = "rural").
- ***C3. Time-interval-based correlation*** is a special kind of property-based correlation. A timestamp is attached to an event and a corresponding label is assigned to the event if the event happened within a given interval. E.g. all events that happen in July 2006 could be grouped together (e.g. label = "07/2006") as opposed to those happening in August (label = "08/2006").

Function-based correlation can be formalized in the following way: Let $Label$ be the set of all labels and $F \subseteq \{f \mid f : E \to Label\}$ a set of partial functions assigning labels to an event. Then the set of sets of correlated events is

$$\{C \subseteq E \mid \exists l \in Label\ (\forall e \in E\ [\exists f \in F\ (l = f(e)) \leftrightarrow e \in C])\}$$

As an extension to function-based correlation relationships between the labels can be considered ($R_L \subseteq Label \times Label$). E.g. we could assume a hierarchical order of keys where several keys have a common super-key. In this case events could be grouped according to their keys attached as well as according to some super-key higher up in the hierarchy. Let us assume e.g. a set of line items that all belong to the same order. In this example events could be grouped according to the line item ID or according to the order ID.

In WS-CDL channel instances can have several identities that are used for correlation. Identities are determined by one or several tokens (keys) and corresponding to labels in our formalization. If two identities l_1, l_2 share a common key, the corresponding labels are related ($(l_1, l_2) \in R_L$).

The formalization given above uses one set of labels. However, in practice we would distinguish between different types of labels, e.g. intervals, product groups, locations.

3.2 Chained Correlation

The basic idea of chained correlation is that we can identify relationships between two events that have to be correlated (grouped together). This relationship

might be explicitly captured in an event's attributes or might be indirectly retrieved by comparing attribute values of two events. Starting from these binary relationships we can build chains of events that belong to the same group.

Since we assume that grouping events to process instances will mostly be done by using unique identifiers, chained correlation becomes important mostly for identifying conversations within our framework. In the case of conversations we especially look at the relationships between message exchanges.

- ***C4. Reference-based correlation***. Two events are correlated, if the second event (in chronological order) contains a reference to the first event. Specifically, this means that if there is some way of extracting a datum from the second event (by applying a function) that is equal to another datum contained in the first event. This datum therefore acts as a message identifier, and the second message refers to this message identifier in some way.
- ***C5. Moving time-window correlation***. Two events involving the same actor are related if they both have the same value for a given function (like in function-based correlation) and they occur within a given duration of one another (e.g. 2 hours). There might be chains of events where the time passed between the first and last event might be very long and others where this time is rather short.

Chained correlation can be formalized in the following way: Let $R \subseteq E \times E$ be the relations between two events that have to be grouped together. Then the set of sets of correlated events is

$$\{C \subseteq E \mid \forall e_1 \in C, e_2 \in E \; [e_1 \; R^* \; e_2 \leftrightarrow e_2 \in C]\}$$

3.3 Aggregation Functions

Sometimes only a limited number of events are grouped together although according to function-based or chaining correlation mechanisms more events would fulfill the criteria to be part of the group. E.g. only a maximum number of 10 items are to be shipped together in one container. More items are requested to be shipped and might have the same destination or arrive timely according to the defined moving time window.

For this additional aggregation of events, special boolean functions agg are defined over sets of correlated events ($agg : \wp(E) \rightarrow \{true, false\}$).

4 Conversation Patterns

The Service Interaction Patterns already describe some of the most recurrent interaction scenarios *within* one conversation. The following patterns focus on relationships *between* different conversations.

4.1 C6. Conversation Overlap

Some interactions belong to two or more conversations. Each conversation also contains interactions that are not part of the others.

E.g. during a conversation centering around delivery of goods a payment notice is exchanged. This payment notice is the starting point for a conversation centering around the payment.

Two conversations $C_1, C_2 \in Conv$ overlap if $C_1 \cap C_2 \neq \emptyset \wedge C_1 \setminus C_2 \neq \emptyset \wedge C_2 \setminus C_1 \neq \emptyset$.

4.2 C7. Hierarchical Conversation

Several sub-conversations are spawned off and merged in a conversation. The number of sub-conversations might only be known at runtime.

E.g. as part of a logistics contract negotiation between a dairy producer and a supermarket chain a set of shippers are to be selected for transporting goods from the producer to the various intermediate warehouses of the chain. Therefore, negotiation conversations are started between the chain and each potential available shipper.

A conversation $C_1 \in Conv$ has two sub-conversations $C_2, C_3 \in Conv$ if $\exists C_p \in Conv\ (C_1, C_2, C_3 \subset C_p \wedge \forall e_2 \in C_2, e_3 \in C_3\ [\exists e_{11}, e_{12} \in C_1\ (e_{11} < e_2 \wedge e_{11} < e_3 \wedge e_2 < e_{12} \wedge e_3 < e_{12})])$

4.3 C8. Fork

A conversation is split into several conversations and is not merged later on. The number of conversations that are spawned off might only be known at runtime.

E.g. an order is placed and the different line items are processed in parallel.

A split from a conversation $C_1 \in Conv$ into the two conversations $C_2, C_3 \in Conv$ is given if $\exists C_p \in Conv\ (C_1, C_2, C_3 \subset C_p \wedge \forall e_1 \in C_1, e_2 \in C_2, e_3 \in C_3\ [e_1 < e_3 \wedge e_1 < e_2])$

4.4 C9. Join

Several conversations that do not originate from the same fork are merged into one conversation. The number of conversations that are merged might only be known at runtime.

E.g. several orders arriving within one week are merged into a batch order.

A join between two conversations $C_1, C_2 \in Conv$ into one conversation $C_3 \in Conv$ is given if $\exists C_p \in Conv\ (C_1, C_2, C_3 \subset C_p \wedge \forall e_1 \in C_1, e_2 \in C_2, e_3 \in C_3\ [e_1 < e_3 \wedge e_2 < e_3])$

4.5 C10. Refactor

A set of conversations is refactored to another set of conversations. The numbers of conversations that are merged and spawned off might only be known at runtime.

E.g. goods shipped in containers on different ships have reached a harbor where they are reordered into trucks with different destinations.

This pattern generalizes Fork and Join.

A refactoring from two conversations $C_1, C_2 \in Conv$ into the two conversations $C_3, C_4 \in Conv$ is given if $\exists C_p \in Conv\ (C_1, C_2, C_3, C_4 \subset C_p \wedge \forall e_1 \in C_1, e_2 \in C_2, e_3 \in C_3, e_4 \in C_4\ [e_1 < e_3 \wedge e_1 < e_4 \wedge e_2 < e_3 \wedge e_2 < e_4])$

5 Process Instance to Conversation Relationships

The correlation mechanisms already describe how to get to event groupings following the notions of conversations and process instances. We assumed so far that conversations and process instances are orthogonal concepts and that groupings can be done independently from each other. This is only partly true. The normal case is that a process instance is involved in one or several conversations and according to which conversation an event belongs to the event is assigned to a particular process instance. Or it is the other way round that an event belonging to the same process instance like a previous event might be assigned to the same conversation.

For clarifying this situation we describe the most important relationships between process instances and conversations. For the first time we use the notion of actors that are part of the framework. We assume that a process instance is executed by exactly one actor and therefore introduce the auxiliary relation $\approx \in \wp(AE) \times \wp(AE)$ where $p_1 \approx p_2$ means that the process instances p_1 and p_2 are executed by the same actor.

Furthermore, we introduce the auxiliary relation $\diamond \subseteq \wp(CE) \times \wp(AE)$ indicating that at least one event in a conversation C is causally related to at least one event in a process instance p. $\diamond = \{(C, p) \in \wp(CE) \times \wp(AE) \mid \exists e_1 \in C\ e_2 \in p\ (e_1 <_c e_2 \vee e_2 <_c e_1)\}$.

5.1 C11. One Process Instance – One Conversation

A process instance is involved in exactly one conversation and there is no other process instance involved in it and executed by the same actor.

E.g. a purchase order is handled within one process instance.

A one-to-one mapping for a process instance $p \in PI$ to conversation $C \in Conv$ is given if

$$p \diamond C \wedge \forall q \in PI\ [(p \neq q \wedge p \approx q) \rightarrow \neg q \diamond C]\ \wedge \forall D \in Conv\ [C \neq D \rightarrow \neg p \diamond D]$$

5.2 C12. Many Process Instances – One Conversation

Several process instances executed by the same actor are involved in the same conversation.

E.g. an insurance claim is handed over from the claim management department to the financial department. The different departments have individual process instances to handle the case.

A many-to-one mapping for a set of process instances $PI' \subseteq PI$ to conversation $C \in Conv$ is given if

$$\forall p_1, p_2 \in PI' \; [p_1 \approx p_2] \wedge \forall p \in PI' \; [p \diamond C]$$

5.3 C13. One Process Instance – Many Conversations

One process instance is involved in many conversations.

E.g. a seller negotiates with different shippers about shipment conditions for certain goods. The shipper offering the best conditions is selected before shipment can begin.

A one-to-many mapping for a process instance $p \subseteq PI$ to a set of conversations $Conv' \in Conv$ is given if

$$\forall C \in Conv' \; [p \diamond C]$$

We can refine this pattern by looking at the relationship between individual sub-process instances (threads) and the conversations. $p_1 \in PI$ is a sub-process instance of $p_2 \in PI$ if all events in p_1 are contained in p_2: $p_1 \subseteq p_2$. Having identified all sub-processes instances we can then analyze if they conform to one of the three mapping patterns.

5.4 C14. Initiator Role

A process instance has the role of the initiator of a conversation if the conversation is started within the process instance.

E.g. a buyer places a purchase order and triggers a conversation concerning the negotiation about the price.

A process instance $p \in PI$ is an initiator of a conversation $C \in Conv$ if $\exists e_1 \in p \; e_2 \in C \; (e_1 <_c e_2 \wedge \neg \exists f \in C \; (f < e_2))$.

5.5 C15. Follower Role

A process instance has the role of a follower (or responder) in a conversation it participates in if the conversation was created within another process instance. The process instance might be created because of a message received in the conversation.

E.g. a purchase order comes in and is processed in a new process instance.

A process instance $p \in PI$ is a follower in a conversation $C \in Conv$ if $\neg \exists e_1 \in p \; e_2 \in C \; (e_1 <_c e_2 \wedge \neg \exists f \in C \; (f < e_2))$.

A process instance is created because of a message in the conversation if $\exists e_1 \in p\ e_2 \in C\ (e_2 <_c e_1 \wedge \neg\exists g \in p \setminus C\ (g < e_1))$.

This second case can be implemented in BPEL by having a receive activity at the beginning of the process. A new process instance is then created as soon as a message of the specified type arrives.

5.6 C16. Leave Conversation

A process instance decides to no longer take part in a conversation.

E.g.

For formalizing this pattern we need to introduce the notion of action event types and conversation types. Functions $AET : AE \rightarrow Type$ and $CT : \wp(CE) \rightarrow Type$ assign a type to each action event and conversation. Leave Conversation is given if $leave \in Type$ is the event type corresponding to leave actions and $lc \in Type$ is the type of conversation that is to be left and for all possible process instances p: $\neg\exists e_1, e_2 \in p\ e_3 \in CE\ (AET(e_1) = leave \wedge e_1 < e_2 \wedge CT(e_3) = lc \wedge e_3 < e_2)$.

5.7 C17. Multiple Consumption

A communication event is consumed several times by one or many process instances.

E.g. an account detail change is requested by a supplier and immediately processed. As part of a more complex fraud pattern this request leads to investigating potential fraud.

A communication event $c \in CE$ is consumed several times if $\|\{e \in AE \mid c <_c e\}\| > 1$.

5.8 C18. Atomic Consumption

One action event is caused by several communication actions.

E.g. a new shipment is started as soon as 500 items with the same destination arrive.

A transactional consumption of a set of communication actions $C \in \wp(CE)$ has occurred if $\exists e \in AE\ (\forall c \in C\ [c <_c e])$.

6 Assessment of BPEL 1.1 and 2.0

Table 1 summarizes the support for Correlation Patterns in BPEL 1.1 and 2.0 where "+" indicates direct support for a pattern, "+/–" partial support and "–" no support.

In the context of contemporary web service standards and middleware reference-based correlation can be realized in at least two ways:

- When using SOAP in conjunction with WS-Addressing, each message contains an identifier (*messageID* header) and may refer to a previous message through the *relatesTo* header. If we assume that these addressing headers are used to relate messages belonging to the same service conversation in a chained manner, it becomes possible to group a raw service log containing all the messages sent or received by a service into traces corresponding to service conversations. The method is applicable when using Oracle BPEL as well as various other web service middleware supporting the WS-Addressing standard. Note however web service middleware supporting WS-Addressing may use the *replyTo* header to correlate messages as opposed to the *relatesTo*. Specifically, the *replyTo* header of the a given message (say M) may contain an URI uniquely identifying the message in question. Subsequently, when another message M' of the opposite directionality is observed that has the same URI this time in the *To* header, M and M' can be correlated.
- The second method is based on the identification of properties that a message has in common with another message belonging to the same service conversation. In BPEL, properties shared by messages belonging to the same service conversation are captured as *correlation sets*. A correlation set can be seen as a function that maps a message to a value of some type. Correlation sets are associated with communication actions. When a message is received which has the same value for a correlation set as the value of a message previously sent by a running service conversation, the message in question is associated with this conversation. This allows one to map messages to service conversations, except for those messages that initialize a correlation set, that is, those messages that start a new conversation. Assuming that in the BPEL abstract process of a service only the initial actions of the protocol initialize correlation sets, and all other actions refer to the same correlation sets as the initial action, each message produced or consumed by the service can be mapped to a service conversation as follows: The full message log is scanned in chronological order. A message is either related to a new service conversation if it corresponds to a communication action that initializes a correlation set, or related to a previously identified service conversation if the values of its correlation set match those of a message sent by the previous service conversation.

One source of limitation of BPEL with respect to correlation is the fact that every message arriving at a port is eagerly correlated to a process instance. In other words, when a message addressed to a Web service is received by the BPEL engine, its headers and contents are inspected and the message is either: (i) assigned to an existing process instance; (ii) used as a basis to create a new process instance; or (iii) rejected. This model is not suitable to capture scenarios where correlation can not be determined on a per-message basis, as in the case of the atomic consumption pattern. Consider for example the following example:

> A shipment aggregation service receives shipment requests from multiple customers, and where possible, aggregates them into a single route. When the service receives a shipment request, two scenarios are possible

depending on the destination (e.g. town or suburb): (i) if there is no pending request for the same destination, a new bundle for this destination is created; otherwise, the request is assigned to the existing bundle for that destination. When there are more than a given amount of shipment requests with the same destination, the corresponding bundle is closed and a delivery route is assigned to it. Subsequent messages to the same destination are then assigned to another bundle. If a bundle has been open for more than a given time window, it is escalated to a human operator. Thus, shipment requests are aggregated in bundles based on their destination, until a bundle either reaches a given size (e.g. 10 requests per bundle) or a given age (e.g. 4 hours).

In this scenario, when a shipment request is received and no existing request for that destination is awaiting correlation, the message is buffered. It is only later, once the timeout has expired or the threshold has been reached, that a process instance is created to deal with either that request alone, or a combination of requests with the same destination.

7 Related Work

At least two programming languages for Web Service developments propose alternative correlation mechanisms to BPEL's one: XL [7] and GPSL [8]. XL directly supports the concept of conversation. Conversations are identified by unique URIs that are included in a SOAP header (similar to WS-Addressing "relatesTo" header). Conversation patterns define when should new conversation URIs be created versus when should existing conversation URIs be reused. With respect to BPEL, XL adds a concept of *conversation timer* which can deal with our time-interval-based correlation pattern. A conversation timer is armed when a service receives the first message related to a conversation: If a message is received by the service after the timeout, this message is treated as part of a new conversation. Arguably, one can achieve a similar effect in BPEL 2.0 using scoped correlation sets combined with alarms and faults, but this would require convoluted code. On the other hand, XL still suffers from the same limitations as BPEL when it comes to dealing with the multiple consumption and atomic consumption patterns.

GPSL [8] on the other hand relies on the concept of *join pattern* to capture correlation scenarios such as the shipment aggregation service above. A join pattern is defined as a conjunction of message channels and a filtering condition: when messages are received over a channel they are stored in a buffer until there is a join pattern that can consume them. For a join pattern to fire, there must be a combination of messages (one per channel in the join pattern) which satisfies the filter. This feature corresponds to the "atomic consumption" pattern. Timeouts are conceptually treated as messages coming from a "timer service", thus enabling time-interval-based correlation. Also, GPSL deals with "multiple consumption" by allowing a service to send (or re-send) a message to itself: so once

a message is consumed, the service can put it back again in the corresponding channel.

Concepts similar to join patterns have been considered in the context of complex event processing [4], where they are called *event patterns*. IBM's Active Correlation Technology [9] for example, provides a rule language to capture event patterns such as "more than four events of a given type happen in a sliding window of 30 seconds". Event rule languages can capture arbitrarily complex correlation patterns. But the question that we attempt to answer is: how much of this event correlation technology is needed in SOA?

In the broader context of enterprise applications, the issue of identifying patterns of correlation has been considered in [10]. However, this work only considers reference-based correlation as supported by WS-Addressing.

8 Conclusion and Outlook

In this paper we have introduced a framework for classifying correlation patterns in service-oriented environments. Using this framework we described a set of patterns that can be used for evaluating languages and systems.

The very next step is to assess common and emerging process description languages, such as BPEL, WS-CDL and Let's Dance, and systems such as SAP's ccBPM.

The framework already points into the direction of patterns for describing relationships between different process instances. These patterns have to be identified. Furthermore, the framework allows for extending the scope of correlation patterns into the area of complex event processing and especially event patterns therein. Such work could give an interesting insight into the relationship between fields such as fraud detection and service-oriented architectures.

References

1. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.
2. D. Jordan, J. Evdemon et al. *Web Services Business Process Execution Language Version 2.0* Public Review Draft Draft. OASIS WS-BPEL Technical Committee, August 2006. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf/
3. W3C. *Web Services Addressing (WS-Addressing).* http://www.w3.org/Submission/ws-addressing/
4. D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley, 2001
5. A. Barros, M. Dumas, and A. H.M. ter Hofstede. Service Interactions Patterns. In *Proceedings of the 3rd International Conference on Business Process Management (BPM)*, Nancy, France, September 2005. Springer Verlag, pp. 302-218.
6. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros. *Workflow Patterns*. Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.

7. D. Florescu, A. Grnhagen, D. Kossmann. "XL: an XML programming language for Web service specification and composition" *Computer Networks* 42(5):641–660, 2003.
8. D. Cooney, M. Dumas, and P. Roe. "GPSL: A Programming Language for Service Implementation" *In Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Vienna, Austria, March 2006. Springer Verlag, pp. 3–17.
9. A. Biazetti and K. Gajda. *Achieving complex event processing with Active Correlation Technology.* November 2005. http://www-128.ibm.com/developerworks/autonomic/library/ac-acact/
10. G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003
11. W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold and J.F. Morar. "Web Services Navigator: Visualizing the Execution of Web Services" *IBM Systems Journal* 44(4):821-845, 2005.

Correlation Patterns	**1.1**	**2.0**	**Remarks**
C1. Key-based correlation	+	+	Correlation sets with one or several properties
C2. Property-based correlation	–	–	Only equality leads to correlation
C3. Time-interval-based correlation	–	–	Only identifiers are used
C4. Reference-based correlation	+	+	Support if WS-Addressing is used (see below)
C5. Moving time-window correlation	–	–	
C6. Conversation overlap	+	+	Several correlation sets for one invoke / receive
C7. Hierarchical conversation	+/–	+	Only partial support in BPEL 1.1 since the number of sub-conversations have to be known at design-time. In this case a different correlation set can be used for every conversation. In the case of BPEL 2.0 properties can be defined not only on the process level (like in BPEL 1.1) but also on a per scope basis. Combined with forEach constructs different instances of the same correlation set definition are then used for handling the different conversations.
C8. Conversation fork	+/–	+	Similar to hierarchical conversation.
C9. Conversation join	+/–	+	Similar to hierarchical conversation.
C10. Conversation refactor	+/–	+	Similar to hierarchical conversation.
C11. One process instance – one conversation	+	+	
C12. Many process instances – one conversation	+	+	
C13. One process instance – many conversations	+	+	
C14. Initiator	+	+	Initiate="yes" for a correlation set in an invoke and "no" for the following send and receive actions.
C15. Follower	+	+	Initiate="yes" for a correlation set in a receive and "no" for the following send and receive actions.
C16. Leave conversation	–	+	In BPEL 1.1 unsubscription cannot be expressed. Once values are set for a property, a subscription for corresponding messages exists until the process instance terminates. In the case of BPEL 2.0 subscription ends as soon as the scope where a property is defined is left.
C17. Multiple consumption	–	–	A message is routed to exactly one process instance / scope and can be consumed by only one receive action.
C18. Atomic consumption	–	–	Only one message can be consumed at a time.

Table 1. Support for correlation patterns in BPEL 1.1 and 2.0