Correlation Patterns in Service-Oriented Architectures

Alistair Barros[1], Gero Decker[2], Marlon Dumas[3], and 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 from 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. Examples 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 [12]. 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 exchange corresponding to the response to this request. Such simple relations

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 245–259, 2007.

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 [12] for correlating messages have been proposed in WS-Addressing [7]. 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 stock availability requests are caused by the same "purchase order" receive event.

Another upcoming standard, namely WS-BPEL [9], 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: (i) a unified conceptualization of the notions of conversation, process and correlation in terms of message events (Section 2); (ii) a set of formally defined *correlation patterns* that cover a spectrum of correlation scenarios in the context of conversational services (Sections 3, 4 and 5); (iii) an evaluation of the degree of support for these correlation patterns offered by relevant Web service standards (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 [11]. 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. In addition to these *communication events* that capture the externally visible behavior of actors, we consider *action events*, which are records of internal activities or internal faults within an actor, as well as *timeout events*. A message send event is directly caused by an action event that *produces* the message in question, while a message receipt event normally leads to (i.e. causes) an action event that *consumes* the message in question. We postulate the existence of a causal relation between communication events and action events. In addition, we postulate the existence of a causal relation between send events and their corresponding receipt events. Figure 1 illustrates these causal relations.

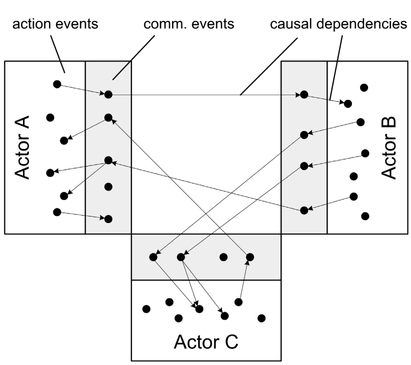

Fig. 1. Action and communication events

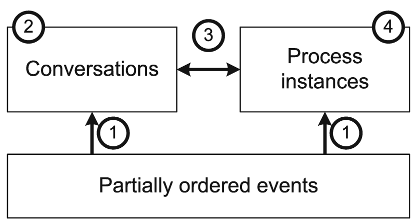

Fig. 2. Framework for classifying correlation patterns

Events can be grouped in different ways, e.g. all events occurring at one particular actor can be grouped together. Since this work deals with event correlation in conversational services, we consider 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 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.

Figure 2 illustrates a framework for classifying correlation patterns. At the bottom there are partially ordered events. We assume that each event has a timestamp, but since events may be recorded by different actors with discrepant clocks, we may not be able to linearly order all events using their timestamps. However, we can use the timestamps to linearly order events recorded by a given actor (assuming a perfect clock within one actor). From there, we can derive a partial order between events recorded by different actors using the relation between message send events and their corresponding receipt events as formalized below. In the case of multiple clocks within one actor due to decomposition into distributed components, causal relations between action events occurring in different components can replace the pure timestamp-based ordering. But for the sake of simplicity, we assume that all events within one actor are totally ordered.

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* are presented in Section 3.
2. Having identified conversations, we can examine how conversations are structured. In previous work we investigated common interaction scenarios between participants within one conversation [2]. The *conversation patterns* in Section 4 present relationships between different conversations.
3. Relationships between conversations and process instances are covered in Section 5.
4. Common patterns of action events within one process instance have been studied in [1]. Additional work is required to identify patterns describing the relationships between different process instances, but this is outside the scope of this paper.

The proposed patterns are formally described 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 necessarily be used to analyze event logs, but rather to assess the capabilities of languages that deal 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 sequel, 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
- $<_t \subseteq E \times E$ 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
- $<$ 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 impose that every message must include a "service instance identifier". Hence, 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 [4] uses IBM's Data Collector to log both the contents and context of SOAP messages. But to capture correlation, the Data Collector inserts a proprietary SOAP header element into every message.

To achieve a general approach to correlation in SOAs, we need to make as few assumptions as possible. In this paper, we assume that message events contain a timestamp and data (i.e. contents), but not necessarily a message identifier. a message identifier as part of their contents, but this is not part of our assumptions. Thus, message event correlation can be performed based on data or based on timestamp. Secondly, we assume that two events can be correlated in the following cases: (1) 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. (2) One event is a cause of the other (directly or transitively), or there is a third event which is a cause (directly or transitively) of both events, or both events are a common cause (directly or transitively) of a third event.

Accordingly, we introduce two categories of correlation mechanisms: *function-based correlation* (case 1 above) and *chained correlation* (case 2). Different flavors of each category are presented, some based on data and others on time. The application of a particular function-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 three correlation mechanisms in this category: the first two deal with correlation based on data, while the third deals with correlation based on time. Strictly speaking, the first two patterns could be merged into a single one (i.e. the second pattern subsumes the first one). However, we treat them separately since, as discussed later, existing standards tend to support the first one but not the second.

C1. Key-based correlation. One or a set of unique identifiers are assigned to an event and all events with at least one common identifier are grouped together. **Example:** a process instance identifier or a conversation identifier is attached to each event. Identifiers can be single values or compositions of several values.

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.
Example: 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. This 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.
Example: 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 $F \subseteq \{f \mid f : E \rightarrow \mathit{Label}\}$ a set of partial functions assigning labels to an event. Then the set of sets of correlated events is $\{C \subseteq E \mid \exists l \in \mathit{Label}\ (\forall e \in E\ [\exists f \in F\ (l = f(e)) \Leftrightarrow e \in C])\}$.

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

As an extension to function-based correlation relationships between the labels can be considered ($R_L \subseteq \mathit{Label} \times \mathit{Label}$). 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.

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. Depending on whether chained correlation is done based on message data or based on time, we can identify two chained correlation mechanisms.

C4. Reference-based correlation. Two events are correlated, if the second event (in chronological order) contains a reference to the first event. 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 as follows: 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. For example, 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.

In this paper, we do not deal with correlation mechanisms that include such maximality requirements. We envisage that the framework could be extended to capture such scenarios by means of an aggregation function *agg* that takes as input a set of correlated events and produces a boolean (i.e. $agg : \wp(E) \rightarrow \{true, false\}$). A correlation scheme could then be constrained to only produce sets of events that satisfy a given aggregation function.

4 Conversation Patterns

The Service Interaction Patterns proposed in [2] describe recurrent interaction scenarios *within* one conversation. The following patterns focus on relationships *between* different conversations.

C6. Conversation Overlap. Some interactions belong to multiple conversations. Each conversation also contains interactions that are not part of others.
Example: during a conversation centering around delivery of goods a payment notice is exchanged. This payment and other payments is the starting point for a conversation centering around the payment.
Two conversations C_1, C_2 overlap if $C_1 \cap C_2 \neq \emptyset \wedge C_1 \setminus C_2 \neq \emptyset \wedge C_2 \setminus C_1 \neq \emptyset$.

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.
Example: 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 $\exists C_p \in Conv\ (C_1, C_2, C_3 \subset C_p \wedge \forall e_2 \in C_2, e_3 \in C_3\ [\exists e_{11}, e_{12} \in C_1\ (e_{11} < e_2 \wedge e_{11} < e_3 \wedge e_2 < e_{12} \wedge e_3 < e_{12})])$.

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.
Example: 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 $C_2, C_3 \in Conv$ occurs if $\exists C_p \in Conv\ (C_1, C_2, C_3 \subset C_p \wedge \forall e_1 \in C_1, e_2 \in C_2, e_3 \in C_3\ [e_1 < e_3 \wedge e_1 < e_2])$.

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.
Example: 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 $C_3 \in Conv$ occurs if $\exists C_p \in Conv\ (C_1, C_2, C_3 \subset C_p \wedge \forall e_1 \in C_1, e_2 \in C_2, e_3 \in C_3\ [e_1 < e_3 \wedge e_2 < e_3])$.

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. This pattern generalizes Fork and Join.
Example: goods shipped in containers on different ships have reached a harbor where they are reordered into trucks with different destinations.
A refactoring from two conversations $C_1, C_2 \in Conv$ into the two conversations $C_3, C_4 \in Conv$ occurs 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

So far, we have considered conversations and process instances separately. Below, we consider relationships between process instances and conversations, as well as relationships between action events and communication events. First, we classify the relationships between process instances and conversations according to multiplicity, and we derive three patterns from there (C11 – C13). Next, we consider relationships between the start and the end of process instances and conversations, and derive another three patterns (C14 – C16). Finally, we consider scenarios that deviate from the usual case whereby one action event is related to one communication event, and derive two more patterns (C17 & C18).

To formalize these patterns, we introduce the notion of *actor*. A process instance is executed by exactly one actor. We rely on a 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. Also, we introduce a 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. $\diamond = \{(C, p) \in \wp(CE) \times \wp(AE) \mid \exists e_1 \in C\ e_2 \in p\ (e_1 <_c e_2 \vee e_2 <_c e_1)\}$.

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.
Example: a purchase order is handled within one process instance.
A one-to-one mapping for a process instance $p \in PI$ to conversation $C \in Conv$ occurs if $p \diamond C \wedge \forall q \in PI\ [(p \neq q \wedge p \approx q) \Rightarrow \neg q \diamond C] \wedge \forall D \in Conv\ [C \neq D \Rightarrow \neg p \diamond D]$.

C12. Many Process Instances – One Conversation. Several process instances executed by the same actor are involved in the same conversation.
Example: 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 $C \in Conv$ occurs if $\forall p_1, p_2 \in PI'\ [p_1 \approx p_2] \wedge \forall p \in PI'\ [p \diamond C]$.

C13. One Process Instance – Many Conversations. One process instance is involved in many conversations.
Example: 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 $Conv' \in Conv$ occurs if $\forall C \in Conv'\ [p \diamond C]$.

C14. Initiate Conversation. A process instance has the role of the initiator of a conversation if the conversation is started within the process instance.
Example: 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 $\exists e_1 \in p\ e_2 \in C\ (e_1 <_c e_2 \wedge \neg \exists f \in C\ (f < e_2))$.

C15. Follow conversation. A process instance p has the role of a follower in a conversation it participates in, if the conversation was created within another process instance. Process instance p may be created because of a message received in the context of the conversation in question.
Example: a shipping order that is part of a multi-party conversation for procuring some products comes in to a shipment process and is processed in a new instance of this process.
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 a conversation if $\exists e_1 \in p\ e_2 \in C\ (e_2 <_c e_1 \wedge \neg\exists g \in p \setminus C\ (g < e_1))$.

C16. Leave Conversation. A process instance decides to no longer take part in a conversation.
Example: a carrier can no longer commit to delivery request and terminates involvement in a shipment contract.

To formalize this pattern we 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 occurs 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: $\neg\exists e_1, e_2 \in p\ e_3 \in CE\ (AET(e_1) = leave \wedge e_1 < e_2 \wedge CT(e_3) = lc \wedge e_3 < e_2)$.

C17. Multiple Consumption. A communication event is consumed (multiple times) by several actions, possibly belonging to different process instances.
Example: 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 is consumed several times if $\|\{e \in AE \mid c <_c e\}\| > 1$.

C18. Atomic Consumption. One action event is caused by several communication actions.
Example: a shipment is started when 500 shipment requests for the same destination are collected (see more detailed example in Section 6).
An atomic 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 BPEL 2.0

In this section, we provide an assessment of BPEL 1.1 and 2.0 specifications for support of the correlation patterns. Since BPEL directly concerns conversational processes, it provides a more comprehensive insight into the capabilities of Web services middleware vis-a-vis of event correlation than standards at lower levels of the WS stack. Table 1 summarizes the assessment, where "+" indicates direct support for a pattern, "+/–" partial support and "–" no support.

The mechanism in BPEL for relating action events to messages (i.e. communication events) is that of *correlation set*. A message can match one or more correlation sets. A BPEL inbound or outbound communication action (e.g. invoke, receive, reply, onMessage, pick) specify one or more correlation sets. These enable the execution engine to determine properties that a message produced or consumed by that action should have.

A process starts a conversation by sending a message and this sending action determines the values of properties in a correlation set, that then serve to identify the communication actions within a process instance that belong to the conversation in question. The conversation is continued by other processes receiving messages containing values of the correlation set. When a message is received which has the same value for a correlation set as the value of a message previously sent as part of a conversation, the message in question is associated with this conversation. Immediately we can see that *key-based correlation* is supported. However, only equality applies so *property-based correlation* is not supported, and no explicit support is available for *time-interval based correlation.*

From a post-mortem perspective, each message produced or consumed by the service can be related to a conversation as follows: the message log is scanned in chronological order, and a message is either related to a new conversation if it corresponds to a communication action that initializes a correlation set, or is related to a previously identified conversation if the values of its correlation set match those of a message sent by the previous service conversation.

Explicit support for *reference-based correlation* is possible when WS-Addressing is used for SOAP message exchange by BPEL processes. In the WS-Addressing standard, a 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 conversation in a chained manner, it is possible to group a service log containing all the messages sent or received by a service into traces corresponding to conversations. Similarly, correlations can be made through the *replyTo* header of a given message (say M), containing an URI uniquely identifying a message in question. When another message M' is observed that has the same URI this time in the *To* header, M and M' can be correlated.

Chaining through sliding windows, addressed in the *moving time-window correlation* pattern, cannot be supported through BPEL. Sliding windows require that events are buffered, however this aspect is left open in the BPEL specifications. A hand-coded solution is to implement buffering through one (continuously running) process instance, but this leads to convoluted code.

Conversation overlap occurs when different correlation sets are used in related message activities of two processes. By way of illustration, consider a process that initiates a conversation through an invoke, which has a corresponding receive in the targeted recipient, having the same correlation set (e.g. PurchaseOrder) as the invoke. Through subsequent message exchanges, reference to a different correlation set is made (e.g. Invoice), providing new data to correlate a different conversation between the two processes.

For conversational structuring, BPEL 1.1 and BPEL 2.0 have a major difference. Both of them allow correlation sets to be defined not only on a per scope basis. However, in BPEL 1.1, either the number of sub-conversations has to be known at design-time (one branch of a "parallel flow" is assign to each conversation), or the conversations in question must be entertained one after the other as opposed to concurrently. In BPEL 2.0, a "parallel foreach" construct, allows an unbounded number of conversations to be entertained concurrently. As such, we can see that *hierarchical conversation*, *conversation fork* and *conversation join* can be fully supported in BPEL 2.0, though only partially supported in 1.1.

The number of correlation sets in and across process instances allows for different conversation multiplicities. Patterns C11–C13 are thus supported. A process can *initiate* a conversation, if in one of its invoke activities, the correlation set's initiate attribute is set to "yes" (with the initiate attribute in the corresponding receive in the participant also set to "yes"). Subsequent message actions of the initiator should then have the initiate attribute be set to "no". Similarly, a process instance *follows* a conversation when one of its receive actions initiates a correlation set, thus signifying that the process instance becomes aware of a conversation. All subsequent actions referring to this correlation set should have the initiate attribute set to "no". For *Leave Conversation*, in BPEL 1.1 unsubscription from a conversation cannot be expressed. Once values are given to a correlation set, a subscription for corresponding messages exists until the process instance terminates. In BPEL 2.0, a subscription ends as soon as the execution of the scope where a correlation set is defined is closed.

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 consumed immediately for instance creation or instance routing, or it is 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 the following scenario: A shipment aggregation service receives shipment requests from multiple customers and aggregates them into bundles. When the service receives a shipment request, it either: (i) creates a new bundle for the shipment's destination if there is no existing bundle for that destination; or (ii) assigns the request to an existing bundle for the same destination. When a bundle reaches a certain size, the corresponding bundle is closed and a delivery route is computed for it. Subsequent messages to the same destination are then assigned to a new bundle. If a bundle has been opened 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) 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 that a process instance is created to deal with either that request alone, or a combination of requests with the same destination. Due to the "per event" nature of its correlation mechanism, BPEL does not support such scenarios involving *multiple consumption* or *atomic consumption*.

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

Correlation Patterns	**BPEL 1.1**	**BPEL 2.0**
C1. Key-based correlation	+	+
C2. Property-based correlation	–	–
C3. Time-interval-based correlation	–	–
C4. Reference-based correlation	+	+
C5. Moving time-window correlation	–	–
C6. Conversation overlap	+	+
C7. Hierarchical conversation	+/–	+
C8. Conversation fork	+/–	+
C9. Conversation join	+/–	+
C10. Conversation refactor	+/–	+
C11. One process instance – one conversation	+	+
C12. Many process instances – one conversation	+	+
C13. One process instance – many conversations	+	+
C14. Initiate conversation	+	+
C15. Follow conversation	+	+
C16. Leave conversation	–	+
C17. Multiple consumption	–	–
C18. Atomic consumption	–	–

The proposed patterns can be used to analyze other languages for Web service implementation, such as the Web Service Choreography Description Language (WS-CDL [10]). WS-CDL uses equality of identity tokens to identify conversations.Therefore, WS-CDL directly supports *key-based correlation* but does not support *property-based correlation*. Like BPEL, WS-CDL can also be used in combination with WS-Addressing leading to direct support for *reference-based correlation*. The timeout attribute for interactions in WS-CDL provides a realization both of *time-interval-based correlation* and of *moving time-window correlation* (note that it is out of the scope of WS-CDL to specify how the individual participants handle the required message buffering). *Conversation overlap* can be realized in WS-CDL through the use of several sets of identity tokens. However, like in BPEL 1.1 there is only partial support for *hierarchical conversation*, *conversation fork* and *conversation join* since WS-CDL provides no construct to capture an unbounded number of branches that execute in parallel. WS-CDL is a language for describing interactions between multiple Web services from a global perspective. It does not deal with the individual behavior of each service, and hence does not have a notion of process instance. Therefore, the patterns from section 5 are irrelevant for WS-CDL.

7 Related Work

Two programming languages for Web service implementation propose alternative correlation mechanisms to BPEL's one: XL [6] and GPSL [5]. 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 suffers from similar limitations as BPEL when it comes to dealing with multiple consumption and atomic consumption.

GPSL 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 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 consuming 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 [11], where they are called *event patterns*. IBM's Active Correlation Technology [3] 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?

Patterns of correlation in enterprise applications are presented in [8]. But this work only considers reference-based correlation as supported by WS-Addressing.

8 Conclusion and Outlook

This paper introduced a framework for classifying and describing correlation scenarios in SOAs, with an emphasis on stateful services that engage in long-running business transactions. Using this framework, we described a set of patterns that can be used to evaluate the correlation mechanisms of standards and tools for service implementation. In particular, we evaluated two successive versions of BPEL and showed that, while the later version supports a larger set of correlation patterns than the earlier, it still does not support certain patterns due to its approach of correlating and consuming messages immediately upon receipt, as well as its inability to deal with time as a factor in determining correlation.

The framework points into the direction of patterns of relationships between process instances. In many scenarios, different process instances compete for the same messages, thus creating dependencies between them. A classification and in-depth study of these dependencies constitutes an avenue for future work. Furthermore, the framework can be extended to cover more sophisticated correlation patterns such as those found in the area of complex event processing. An extended version of the framework could provide a basis for evaluating the correlation mechanisms of languages and systems for event processing in general.

Acknowledgments. The second author conducted this work while visiting SAP Research Centre, Brisbane. The third author is funded by a fellowship co-sponsored by SAP and Queensland Government.

References

1. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. *Workflow Patterns*. Distributed and Parallel Databases, 14(3):5–51, July 2003.
2. A. Barros, M. Dumas, and A. ter Hofstede. Service Interactions Patterns. In *Proceedings of the 3rd International Conference on Business Process Management (BPM)*, Nancy, France, September 2005, pages 65–80. Springer, 2005.
3. A. Biazetti and K. Gajda. *Achieving complex event processing with Active Correlation Technology*. IBM Technical Report, November 2005. http://www-128.ibm.com/developerworks/autonomic/library/ac-acact/
4. 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.
5. 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, pages 3–17. Springer, 2006.
6. D. Florescu, A. Grünhagen, and D. Kossmann. XL: an XML programming language for Web service specification and composition. *Computer Networks* 42(5):641–660, 2003.
7. M. Gudgin, M. Hadley, and T. Rogers (editors). *Web Services Addressing 1.0 – Core.* W3C Recommendation, May 2006. http://www.w3.org/Submission/ws-addressing/
8. G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003
9. D. Jordan and J. Evdemon (editors). *Web Services Business Process Execution Language Version 2.0* Public Review Draft. OASIS WS-BPEL Technical Committee, August 2006. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.pdf/
10. N. Kavantzas, D. Burdett, T. Fletcher, Y. Lafon, and C. Barreto (editors). *Web Services Choreography Definition Language Version 1.0.* W3C Candidate Recommendation, November 2005. http://www.w3.org/TR/ws-cdl-10/
11. D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley, 2001.
12. S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.