SPECIAL ISSUE PAPER

An event view specification approach for Supporting Service process collaboration

Jian Cao1,*, Jie Wang2, Haiyan Zhao3 and Minglu Li1

1Department of Computer Science and Engineering, Shanghai Jiaotong University, 200240 Shanghai, China
2Department of Civil and Environment, Stanford University, CA 94305, USA
3School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, 200093 Shanghai, China

ABSTRACT

Designing and implementing an interoperable and flexible service process collaboration strategy is one of key issues for business to business integrations. To better support service process collaboration, an event view model is proposed, which is composed of a set of event types and their dependency relationships. It provides a general and flexible way to define a public view of a service process model and serves as the basis for defining service process collaboration protocols. In the paper, the basic concepts and a system framework for event-based service process collaboration are first introduced. The definitions of event and the dependency relationships among event types are then presented. Especially, how to identify dependency relationships among composite event types is studied in detail. After discussing the definition of event view and its specifying approach, a procedure for transforming a BPEL process model into an event model and deriving dependencies among events is given. Finally, a case study is presented, and some implementation issues for defining and publishing an event view are discussed. Copyright © 2013 John Wiley & Sons, Ltd.

Received 21 July 2012; Accepted 13 December 2012

KEY WORDS: event view; composite event; process collaboration; service process

1. INTRODUCTION

To support business process automation, embedding a process management component in almost all enterprise software products has been becoming a popular practice [1]. The processes in these software products should be orchestrated to support application integration [2, 3]. Furthermore, with the growth of e-commerce and the trend toward increasing globalization of operations and outsourcing of functions to external providers, there is an emerging need to integrate and coordinate processes that span organizational boundaries [4–6].

To support process collaboration, a distributed integration framework is needed not only for information exchanging among business partners but also for coordinating execution orders of activities coming from separate service processes. Because business collaboration among partners is quite dynamic, this framework should be scalable and able to facilitate connecting to relevant partners dynamically. Recent years, service-oriented computing (SOC) is becoming an enabler for such process collaboration [7]. SOC relies on the technology stack of a service-oriented architecture (SOA) [8].
Although SOA presents some architectural advantages for system integration and process collaboration, it still has its own limitations [9, 10]. EDA is a software architecture pattern promoting the production, detection, consumption of, and reaction to events [11]. When an event happens, those who are interested in this event can be notified. Because events are detected and transferred through some event components, it decouples the client and the services. The fact that EDA can complement SOA, by providing services through triggered events, has already widely recognized by software vendors. A set of specifications that standardize the way Web services interact using events have been released [12, 13]. Some service process engine like Apache ODE can generate events to let outside track what is exactly happening in the engine [14].

Although the combination of SOA and EDA brings a more flexible underlying structure to support process collaboration, we still need some mechanisms to coordinate processes on top of this structure. Therefore, a couple of process specifications such as BPEL [15], WSCI [16], and BPML [17] were defined to describe external transactions and message exchanges between or among business partners. E-business standards, such as ebXML [18] and RosettaNet [19], also provide business process protocols, so do business process specification schema (BPSS) and Partner Interface Processes (PIP) [20].

To specify and enforce a process collaboration protocol relies on process view specifications, which describe the external observable behaviors of each internal process. Obviously, the contents of a process view are different from the original internal process model. This is because organizations need to preserve privacy and autonomy of their processes. Another reason is too much detail information is not necessary for outside partners and they only want to be notified when meaningful status change happens. The information needed by other organizations may also differ from the information directly coming from the original process. For example, partners may want to know whether an order is processed or not. This information cannot be obtained from an inherent order management process directly because there is no single activity that can be exactly called order processing in the model.

To define a process view that can describe external observable behaviors correctly is becoming an important task. However, process specifications cannot solve the problem of how to design correct specifications by themselves.

Several formal process modeling techniques have been provided to address this problem. By applying them, correct process views can be generated from a base process model. These techniques are roughly divided into two classifications: abbreviation and combination. Through abbreviation, some activities are defined as silent activities that are not observable. By renaming activities to silent activities, the desired abstraction can be obtained. This method is often based on process algebras and Petri Nets [21, 22]. In the second method, the process view is derived through the bottom-up combination of activities to provide various levels of abstraction of a process [23].

Although abbreviation and combination are ways to generate process views, sometimes we need to define a new state that cannot be derived by simply combing several activities together. For example, in an order fulfillment process, a delay can be defined to reflect the situation that a received high-priced purchase order is not processed in 3 days. This state cannot be derived by applying abbreviation or combination on the base process model directly.

In this paper, an event view specification approach to support service process collaboration is proposed under the context of a software architecture based on SOA and EDA. In this model, the event view defines external behaviors of the base service process model, and it includes a set of event types and their dependency relationships. Partners who subscribe event types defined in the event view can be notified and hence react to them accordingly when corresponding events happen. At the same time, they will also know what will happen next according to the event dependency information defined in the event view. Comparing with current methods to generate the process view, our event view model is more general and flexible because a composite event can reflect complex state change. In addition, the event-view-based process collaboration framework is easier to implement, and it can be integrated with current EDA structure directly.

The paper is organized as follows. Section 1 serves as an introduction. Section 2 presents the idea of the event-view-based process collaboration model and its supporting system framework. In Section 3, the event model is defined, and dependency relationships among event types are given. In Section 4, the composite event model and how to infer events’ dependency on the basis of dependency relationships among primitive event types are discussed in detail. Section 5 defines the event view
formally and discusses how to specify it. An approach for creating the event view from a BPEL process model is given in Section 6. A case study and relevant implementation issues are presented in Section 7. Finally, Section 8 provides related work, and Section 9 concludes the whole paper.

2. SERVICE PROCESS COLLABORATION BASED ON AN EVENT VIEW

Events are raised when things change. Other components then listen to events and react to them appropriately. Multiple components or partners work together by communicating with each other through sending and receiving events when their internal states change. It is called an event-based collaboration pattern.

Because status changes in a service process instance can be expressed as events, a running service process can expose an event set $ES$ to the outside applications or partners. In this event set, besides events directly detected from an original process model (e.g., the completion of each activity), composite events can also be defined and detected to reflect the complex status changes (e.g., both activity $C$ and $D$ are completed).

Multiple event sets can be defined for one process model, which reflects different business strategies. For example, a general sales business process model is shown in Figure 1 [24]. As an example, two event sets are defined according to different business strategies (RFQ is the abbreviation of Request for Quotation):

<table>
<thead>
<tr>
<th>Event Set 1</th>
<th>Event Set 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. RFQ Received: Activity “receiveRFQ” is completed</td>
<td>1. Received: Activity “receiveRFQ” is completed</td>
</tr>
<tr>
<td>2. RFQ Processing Completed: Activity “prepareQuote” or “prepareRejectRFQ” starts executing</td>
<td>2. Processing Delayed: Activity “checkAvailability” does not start executing within three days after a RFQ received</td>
</tr>
<tr>
<td>3. RFQ Processing Completed: Activity “prepareQuote” or “prepareRejectRFQ” is completed</td>
<td>3. Product Available: Activity “checkAvailability” is completed and condition “goodsAvailable” is true</td>
</tr>
<tr>
<td>5. Order Processing Completed: Activity “processOrder” is completed</td>
<td>5. Quote Sent: Activity “sendQuote” is completed</td>
</tr>
<tr>
<td>6. Reject RFQ Sent: Activity “sendRejectRFQ” is completed</td>
<td>6. Reject RFQ Sent: Activity “sendRejectRFQ” is completed</td>
</tr>
<tr>
<td>8. Order Processing Completed: Activity “processOrder” is completed</td>
<td>8. Order Processing Completed: Activity “processOrder” is completed</td>
</tr>
</tbody>
</table>

In the first event set, even when a customer receives the notification of RFQ processing begins, the company still can change their decisions. In the second set, after event “RFQ Rejected” is sent to the customer, the company has no chance to change the decision in general case. The customer can try to find another provider as soon as possible. Therefore, different event sets mean different business risks for partners.

The dependency information among events is also important because partners need to know what will come up after some events have happened so that they can define their ordered reactions. For example, if the relationship that after an event “RFQ Rejected” happens event “RejectRFQ Sent” must happen later is announced, the partners can regard either the event “RFQ Rejected” or “RejectRFQ Sent” as the rejection notification. However, if the company does not provide such causal information, it means that “RejectRFQ Sent” will not necessarily happen after “RFQ Rejected” happened. Therefore, the partner cannot wait for the occurrence of “RejectRFQ Sent” for making further decision. In some cases, the dependency among these events need not be specified because they are obvious. But in most cases, these relationships should be specified explicitly.

Selected event types together with their dependency relationships can be published as an event view. It serves as the basis for service process collaboration, which is shown in Figure 2. Process A can
publish events $E^A_1$ and $E^A_2$, while process $B$ publishes event $E^B_1$. A process can subscribe events from other processes. For example, process $A$ subscribes $E^B_1$ from process $B$, and process $A$ subscribes $E^A_1$ and $E^A_2$ from process $A$. These two processes are synchronized by receiving event notifications and then executing related activities correspondingly.

A publish/subscribe network can be taken advantage of as a communication layer to support event-view-based process collaboration (see Figure 3), where event brokers are linked together for exchanging event information.

For each partners involving in collaboration, they often have some process centered applications deployed, for example, a workflow engine or an ERP system. These applications will emit events to
event brokers or consume events obtained from event brokers. To support the integration of heterogeneous applications with an event broker, dedicate adapters should be developed in terms of application types. Because common interfaces of event brokers are defined, the adapters that conform to these interfaces can be easily developed.

The event broker is the connector and coordinator among process centered applications. It can be divided into two parts, that is, an event publication and subscription interface and an event manager. The event publication and subscription interface is responsible for sending and receiving events according to the registration information. The main function of an event manager is event detection, which includes to detect primitive events and to detect composite events.

The framework presented previously is a conceptual one. It can be realized by combing existing software packages and components to be developed. The publish/subscribe network can be based on a wide area Pub/Sub network such as Siena [25], or even some Java Message Service (JMS) API [26] conformed middlewares when it is applied within an enterprise. The composite event detector can be based on some event engines, such as Esper [27].

3. A SEMANTIC EVENT MODEL

An event is a description of status changes of a certain object that is of interest to the outside world. In an event system, an event can be classified into a primitive event or a composite event base on the granularity of the event. Because an event describes a status change and the status is often defined in terms of parameters, an event expression usually contains a set of parameters.

Traditionally, an event parameter is treated as the local information defined for an event source itself, not being shared across the whole application domain. Obviously, the lack of information sharing will result in event communication inefficiency and furthermore the infeasibility of intelligently processing of events. In this section, we define a semantic primitive event model to overcome the limitations.

3.1. Semantic primitive events

A primitive event can be detected by the system directly. The system monitors status of environment and applications continuously, periodically or notified directly by event sources. As soon as being detected, the primitive event is recorded in the event history. An occurrence of an event type is called an event instance.

A primitive event type $E$ is defined as $E = \langle \text{Name}_E, \text{Para}_S \rangle$, where

- $\text{Name}_E$ is the name of the event type, which is a unique identifier for this event type;
- $\text{Para}_S$ defines the information that this type of event will have, and it can be further represented as $\text{Para}_S = \{p_1, p_2, \ldots, p_n\}$, where $p_i$ is a parameter. A function $\text{Para}_S(E)$ is applied to obtain the parameter set of event type $E$. There are two special parameters, $\text{OT}$ and $\text{OI}$, in all event type definitions. $\text{OT}$ is the occurrence time of an event. We assume a primitive event is an instantaneous, atomic occurrence of an interest at a point in time. In fact, the actual occurrence time of the event cannot be known, and we can only infer it on the basis of some assumptions. $\text{OI}$ is called occurrence index, which is used to indicate how many instances of $E$ have happened.

Two examples of primitive event types are as follows:

$$E_{\text{OrderSubmission}} = \langle \"OrderSubmission\", \{\text{Order, Customer, \ldots}\} \rangle$$
$$E_{\text{ActivityLifecycleEvent}} = \langle \"ActivityLifeCycleEvent\", \{\text{ProcessID, InstanceID, ActivityID, fromActivityStatus, toActivityStatus, \ldots}\} \rangle$$

Primitive event types can be organized in a taxonomy (Figure 4). Process events represent status changes of process related objects during the process execution, which will be discussed in detail in Section 6. Business object events describe the status changes of business objects in a process. Service operation event reflects the invocation status of a service operation.

The parameters in the event type definition are defined as terms in ontology. Ontology is an explicit specification of a conceptualization, and its importance has been well widely recognized [28]. In our
model, the domain ontology is structured as a set of individual generalization hierarchy terminology trees, with the more abstract concepts of the ontology forming the root terms of which other terms are specified. As a term, it often has a set of attributes. If a term $a$ inherits from a term $b$, then $a$ specializes $b$, and it is denoted as $a \in SP_b(b)$. Accordingly, $b$ generalizes $a$. The relationships of specialization (or generalization) are transitive. For example, if $c \in SP_a(a)$ and $a \in SP_b(b)$, then $c \in SP_b(b)$. For two sets $A$ and $B$, if $\forall b_1 \in B$, $\exists a_1 \in A$, $a_1 = b_1$ or $a_1 \in SP_b(b_1)$, then $A$ specializes $B$, and it is denoted as $A \subseteq SP_b(B)$. If $A \neq B$, then it is denoted as $A \subset SP_b(B)$.

For two event types $E_a$ and $E_b$, if $ParaSE_{E_b} \subseteq SP(SE_{E_a})$, we call event type $E_b$ specializes event type $E_a$, and it is denoted as $E_b \in SP_a(E_a)$. Accordingly, $E_a$ generalizes $E_b$.

On the basis of the primitive event type definition, we can define the restricted event type, which adds a filter to it. A filter consisting of a single atomic predicate is a simple filter. Filters that are derived from simple filters by combining them with conjunction or disjunction operators are compound filters. An attribute filter is a simple filter that imposes a constraint on the value of a single attribute (e.g., $Order.Worth >$ $1M$). It is defined as a triple $f_i = (a_i, Opi_i, Ci)$, where $a_i$ is an attribute, $Opi_i$ is a test operator, and $Ci$ is a set of constants that may be empty.

A restricted primitive event type can be denoted as $E' = <Name_{E'}, Name_{E}, ParaS, filtSE>$, where $Name_{E'}$ is the restricted event type name, $Name_{E}$ is the name of its attached even type, $ParaS$ is the parameter set inherited from its attached event type, and $filtSE$ is the filter. For example, we can define a restricted event type $E'_{\text{High-pricedOrderSubmission}} = <\text{"High-pricedOrderSubmission"}, \text{"OrderSubmission"}, \{\text{Order, Customer, ...}, f_1: \text{Order.Worth} >$ $1M\}>$. Another example is $E'_{\text{ActivityCompleted}} = <\text{"ActivityCompleted"}, \text{"ActivityLifecycleEvent"}, \{\text{...}, \text{fromActivityStatus} = \text{"Executed" OR \"Waiting\"}) \text{toActivityStatus = \"Completed\"}>$, which can be fired when an activity is completed.

The detection of the occurrence of a restricted event type is triggered by the detection of occurrence of its attached event type. For example, when an event "OrderSubmission" occurs, the system will detect the occurrence of $E'_{\text{High-pricedOrderSubmission}}$, and if $f_1$ is evaluated to be TRUE, then $E'_{\text{High-pricedOrderSubmission}}$ occurs.

The restricted event type is a specialization of its attached event type, that is, $<Name_{E'}, Name_{E}, ParaS, filtSE > \in SP_b(<Name_{E}, ParaS>)$. Furthermore, for two restricted event types

$E'_a = <Name_{E'a}, Name_{E}, ParaS, filtSE_a>$ and $E'_b = <Name_{E'b}, Name_{E}, ParaS, filtSE_b>$:

If $filtSE_a = \text{TRUE} \Rightarrow filtSE_b = \text{TRUE}$, then $E'_d \in SP_a(E'_d)$.

For example, if we defined $E'_{\text{VeryHighPricedOrderSubmission}} = <\text{"VeryHighPricedOrderSubmission"}, \text{"OrderSubmission"}, \{\text{Order, Customer, ...}, f_1: \text{Order.Worth} >$ $10M\}>$, then $E'_{\text{VeryHighPricedOrderSubmission}} \in SP_a(E'_{\text{High-pricedOrderSubmission}})$.

### 3.2. Dependency relationships between two primitive event types

In an application domain, the relative orderings of occurrence times of events often follows some rules. For example, only after an order has been placed, it can be processed. There are already some
dependency models presented, and the widely accepted one is causality, which is a modalized precedence relation ("precedes" plus some modal statement) [28–30]. For example, in Snoop, the causality means event \( E_1 \) causally precedes event \( E_2 \) if and only if the existence of \( E_1 \) will definitely cause the occurrence of event \( E_2 \) in the future [31].

For a domain, if our knowledge is complete, then we can enumerate all the causality relationships among events so that the dynamic behavior of the system is deterministic. But due to some reasons [32], we often have no complete knowledge for a domain. Therefore, more often than not the complete causality relationships are not modeled for the domain. To reflect the uncertainties caused by incomplete knowledge or ignorance, we differentiate two aspects of causality into another two dependency types.

3.2.1. The dependency type. A dependency relationship between two event types is denoted as \( DR=\langle \text{DepT}, \text{CorrelS}\rangle \), where \( \text{DepT} \) is the dependency type and \( \text{CorrelS} \) is the correlation set. \( \text{CorrelS}() \) is a function to obtain the correlation set of the dependency relationship.

The dependency relationship types include

1. **Following Relationship**: For two event types \( E_a \) and \( E_b \), when \( E_a \) occurs, \( E_b \) must occur in the future. We call \( E_b \) follows \( E_a \), and it is denoted as \( E_a \rightarrow F E_b \).

2. **Relying on Relationship**: For two event types \( E_a \) and \( E_b \), if \( E_b \) occurs, \( E_a \) must have already occurred in the past. We call \( E_b \) relies on \( E_a \), and it is denoted as \( E_a \rightarrow R E_b \).

3. **Causality Relationship**: If two event types satisfy both following and relying on relationships, that is, \( (E_a \rightarrow F E_b) \land (E_a \rightarrow R E_b) \) holds, we call event \( E_a \) causally precedes event \( E_b \), and it is denoted as \( E_a \rightarrow C E_b \).

We use \( \text{DRT}(E_a, E_b) \) as the function to obtain the dependency type between two event types \( E_a \) and \( E_b \), that is, to check whether \( E_a \rightarrow F E_b \), \( E_a \rightarrow R E_b \), or \( E_a \rightarrow C E_b \) holds or not.

Obviously, the dependency relationship is transitive, that is,

\[
\text{Rule 3.1} : (E_a \rightarrow F E_b) \land (E_b \rightarrow R E_c) \Rightarrow (E_a \rightarrow R E_c) \\
\text{Rule 3.2} : (E_a \rightarrow R E_b) \land (E_b \rightarrow R E_c) \Rightarrow (E_a \rightarrow R E_c) \\
\text{Rule 3.3} : (E_a \rightarrow C E_b) \land (E_b \rightarrow C E_c) \Rightarrow (E_a \rightarrow C E_c)
\]

Typically, for a domain, multiple dependency relationships will be defined for some event type pairs. These dependency relationships should be rational; that is, they should be logically consistent with the correct system behaviors. For this reason, we add two explanations on the dependency relationships. For a set of event dependency relationships,

1. If both \( E_a \rightarrow F E_c \) and \( E_b \rightarrow F E_c \) appear in the set: it means either \( E_a \) or \( E_b \) leads to the occurrence of \( E_c \) in the future independently.

2. If both \( E_a \rightarrow R E_c \) and \( E_b \rightarrow R E_c \) appear in the set: it means when \( E_c \) occurs, both \( E_a \) and \( E_b \) must have already occurred in the past.

3.2.2. The correlation set for the dependency relationship. Simply defining the dependency type is not sufficient for accurately describing the dependency between two event types. For example, if we have a following causality relationship between two event types:

\[
E_{\text{OrderSubmission}} \rightarrow C E_{\text{ApprovalRequest}}
\]

And we have four events occurred along with the time dimension (see Figure 5):
When we observe the occurrences of \( e_1 \) and \( e_4 \), they conform to the dependency relationship as defined. But according to our knowledge, they have no causal relationship because they belong to different process instances.

The correlation set defines the constraints on two parameters coming from the definitions of two event types. The simplest constraint is common parameter constraint, which defines common parameters that will have the same values for event instances of these two event types. For example, a correlation set for event type \( E_{\text{OrderSubmission}} \) and \( E_{\text{ApprovalRequest}} \) can be \{\text{order}\}. But we can also define more complex constraints, for example, \( OI_a = OI_b + 1 \). It means a dependency relationship exists between a pair of event instances only if the occurrence index of the event instance of \( E_a \) is greater by one than the occurrence index of the event instance of \( E_b \). Unless otherwise indicated, the correlation set only includes common parameter constraints and \{\text{OI}\} is a default common parameter.

Suppose there are \( DR(E_a, E_b) = \langle \text{DepT}, \text{CorrelS}_{ab} \rangle \) and \( DR(E_b, E_c) = \langle \text{DepT}, \text{CorrelS}_{bc} \rangle \), we can derive that \( DR(E_a, E_c) = \langle \text{DepT}, \text{CorrelS}_{ab} \cup \text{CorrelS}_{bc} \rangle \). \text{CorrelS}(E_a, E_b) \) is a function to obtain the correlation set for the dependency relationship between \( E_a \) and \( E_b \).

### 3.2.3. Restricted dependency relationships.

Sometimes, the existence of a dependency relationship relies on the fact that some requirements can be satisfied. For this kind of relationship, we call it a restricted dependency relationship and denote it as \( DR = \langle \text{DepT}, \text{CorrelS}_{ab} \rangle \), where \( \text{filtS}_{ab} \) is a filter, and only when it is evaluated to be TRUE, the dependency relationship can be maintained.

We can denote a restricted dependency relationship as \( E_a \xrightarrow{(\text{DepT}, \text{filtS}_R)} E_b \).

When the transitive relationship is applied to the restricted dependency types, we have a following rule:

\[
\text{Rule 3.4 : } E_a \xrightarrow{(\text{DepT}, \text{filtS}_{ab})} E_b, E_b \xrightarrow{(\text{DepT}, \text{filtS}_{bc})} E_c \Rightarrow E_a \xrightarrow{(\text{DepT}, \text{filtS}_{ab} \cup \text{filtS}_{bc})} E_c
\]

We also have following rules when there are multiple restricted dependency types in the domain:

\[
\text{Rule 3.5 : } E_a \xrightarrow{(\text{DepT}, \text{filtS}_{R1})} E_b, E_a \xrightarrow{(\text{DepT}, \text{filtS}_{R2})} E_b \Rightarrow E_a \xrightarrow{(\text{DepT}, \text{filtS}_{R1} \lor \text{filtS}_{R2})} E_b
\]

For a restricted dependency type, when we know the value of a filter, the restricted dependency type becomes deterministic.

### 4. COMPOSITE EVENT TYPE AND ITS DEPENDENCY MODEL

#### 4.1. Composite event operators

To describe the complex status change in a domain, primitive event types can be composed through a set of operators. We call these primitive event types the member events of the composite one. There are many operator definitions ranging from simple ones to complex ones in the literatures [31–34], and they are similar in their functionalities to some degree. Typical operators defined in Snoop are as follows [31]:

- Disjunction (\( \lor \)): Disjunction of two event types \( E_1 \) and \( E_2 \), denoted as \( E_1 \lor E_2 \) and happens when \( E_1 \) occurs or \( E_2 \) occurs.
- Sequence (\( ; \)): Sequence of two events \( E_1 \) and \( E_2 \) is a composite event, which is denoted \( E_1 ; E_2 \), and occurs when \( E_2 \) occurs provided \( E_1 \) has already occurred.
Conjunction (Any, All): The conjunction event, denoted \( \text{ANY}(I, E_1, E_2, \ldots, E_n) \) where \( I \leq n \), occurs when any \( I \) events out of the \( n \) events (corresponding to \( n \) distinct events specified) occur, ignoring the order of their occurrence.

A periodic Event Operator (\( A, A^* \)): it allows one to express the occurrence of an aperiodic event bounded by two arbitrary events (for providing an interval). The non-cumulative variation of an aperiodic event is expressed as \( A(E_1, E_2, E_3) \), and the event \( A \) is signaled each time \( E_2 \) occurs during the closed interval defined by the occurrences of \( E_1 \) and \( E_3 \). \( A^*(E_1, E_2, E_3) \) occurs only once when \( E_3 \) occurs and accumulates the parameters for each occurrence of \( E_2 \).

Periodic Event Operator (\( P, P^* \)). It is defined as an event \( E \) that repeats itself within a constant and finite amount of time.

More detail explanations on the aforementioned operators can be found in [31]. Among these operators, aperiodic event operator and periodic event operator are special. Aperiodic event can be regarded as a kind of sequential event. And the event of interest in operators, aperiodic event operator and periodic event operator are special. Aperiodic event can be directly. For example, after both two events of \( \text{Ea} \) and \( \text{Eb} \) occur, another event of \( \text{Ec} \) will occur in the future: \( (E_a \land E_b)^E E_c \).

To check whether this rule is violated or not, we should detect possible causal relationships between \( E_a \) and \( E_b \). We discuss the approach for detecting such relationships in the next section.

4.2. Dependency relationships between two composite event types

Dependency relationships can also exist between two composite event types or between one composite type and one simple event type.

Dependency relationships for composite events can be defined according to the domain knowledge directly. For example, after both two events of \( E_a \) and \( E_b \) occur, another event of \( E_c \) will occur in the future: \( (E_a \land E_b)^E E_c \).

On the other hand, the dependency relationship can also be able to be identified from dependency relationships among member event types, which will be discussed in this section. To explore how to identify the dependency relationship between two composite event types, we firstly discuss how to identify the dependency between a single event type and a composite event type. It can be further divided into two scenarios:

- Conjunction (Any, All): The conjunction event, denoted \( \text{ANY}(I, E_1, E_2, \ldots, E_n) \) where \( I \leq n \), occurs when any \( I \) events out of the \( n \) events (corresponding to \( n \) distinct events specified) occur, ignoring the order of their occurrence.
- A periodic Event Operator (\( A, A^* \)): it allows one to express the occurrence of an aperiodic event bounded by two arbitrary events (for providing an interval). The non-cumulative variation of an aperiodic event is expressed as \( A(E_1, E_2, E_3) \), and the event \( A \) is signaled each time \( E_2 \) occurs during the closed interval defined by the occurrences of \( E_1 \) and \( E_3 \). \( A^*(E_1, E_2, E_3) \) occurs only once when \( E_3 \) occurs and accumulates the parameters for each occurrence of \( E_2 \).

Periodic Event Operator (\( P, P^* \)). It is defined as an event \( E \) that repeats itself within a constant and finite amount of time.

More detail explanations on the aforementioned operators can be found in [31]. Among these operators, aperiodic event operator and periodic event operator are special. Aperiodic event can be regarded as a kind of sequential event. And the event of interest in operators, aperiodic event operator and periodic event operator are special. Aperiodic event can be directly. For example, after both two events of \( \text{Ea} \) and \( \text{Eb} \) occur, another event of \( \text{Ec} \) will occur in the future: \( (E_a \land E_b)^E E_c \).

To check whether this rule is violated or not, we should detect possible causal relationships between \( E_a \) and \( E_b \). We discuss the approach for detecting such relationships in the next section.

4.2. Dependency relationships between two composite event types

Dependency relationships can also exist between two composite event types or between one composite type and one simple event type.

Dependency relationships for composite events can be defined according to the domain knowledge directly. For example, after both two events of \( E_a \) and \( E_b \) occur, another event of \( E_c \) will occur in the future: \( (E_a \land E_b)^E E_c \).

On the other hand, the dependency relationship can also be able to be identified from dependency relationships among member event types, which will be discussed in this section. To explore how to identify the dependency relationship between two composite event types, we firstly discuss how to identify the dependency between a single event type and a composite event type. It can be further divided into two scenarios:
1) A composite event type depends on a single event type

In this scenario, the member event types of a composite event type $E_b$ may have dependencies on a single event type $E_a$ (It can be a primitive one or composite one). We should determine under what kind of conditions $E_b$ depends on $E_a$. We will discuss the rules in terms of different composite operators.

(1) Disjunction ($\vee$)

For $E_b = E_a \cup E_{b_i} \cup \cdots \cup E_{b_n}$, we have

Rule 4.2: $\exists i \in [1, n], E_a \xrightarrow{F} E_{b_i} \Rightarrow E_a \xrightarrow{F} E_b$ with $\text{Correl}(E_a, E_{b_i})$

$= \text{Correl}(E_a, E_{b_i}) \cup \text{Correl}(E_b)$

Rule 4.3: $\forall i \in [1, n], E_a \xrightarrow{R} E_{b_i} \Leftrightarrow E_a \xrightarrow{R} E_b$ with $\text{Correl}(E_a, E_{b_i})$

$= \left( \bigcup_{i=1}^{n} \text{Correl}(E_a, E_{b_i}) \right) \cup \text{Correl}(E_b)$

(2) Sequence ($;$)

For $E_b = E_{b_1} ; E_{b_2} ; \cdots ; E_{b_n}$, we have:

Rule 4.4: $\exists i \in [1, n], E_a \xrightarrow{R} E_{b_i} \Leftrightarrow E_a \xrightarrow{R} E_b$ with $\text{Correl}(E_a, E_{b_i})$

$= \text{Correl}(E_b)$

(3) Conjunction (Any, All)

For $E_b = \text{ANY}(I, E_{b_1} ; E_{b_2} ; \cdots ; E_{b_n})$, we have

Rule 4.5: $\forall j \in [1, m], I \leq m \leq n, E_a \xrightarrow{F} E_{b_j} \Leftrightarrow E_a \xrightarrow{F} E_b$ with $\text{Correl}(E_a, E_{b_j})$

$= \bigcup_{j=1}^{m} \text{Correl}(E_a, E_{b_j}) \cup \text{Correl}(E_b)$

Rule 4.6: $\forall j \in [1, m], m > n - I, E_a \xrightarrow{R} E_{b_j} \Leftrightarrow E_a \xrightarrow{R} E_b$ with $\text{Correl}(E_a, E_{b_j})$

$= \left( \bigcup_{j=1}^{m} \text{Correl}(E_a, E_{b_j}) \right) \cup \text{Correl}(E_b)$

2) A single event type depends on a composite event type

In this scenario, one event type has dependencies on several event types, and these event types are composed.

(1) Disjunction ($\vee$)

For $E_b = E_a \cup E_{a_1} \cup \cdots \cup E_{a_m}$

Rule 4.7: $\exists i \in [1, n], E_a \xrightarrow{F} E_{a_i} \Leftrightarrow E_a \xrightarrow{F} E_b$ with $\text{Correl}(E_a, E_{a_i}) = \text{Correl}(E_a, E_b) \text{Correl}(E_b)$

Rule 4.8: $\forall i \in [1, n], E_a \xrightarrow{R} E_{a_i} \Leftrightarrow E_a \xrightarrow{R} E_b$ with $\text{Correl}(E_a, E_{a_i}) = \bigcup_{i=1}^{n} \text{Correl}(E_a, E_{a_i}) \text{Correl}(E_b)$

(2) Sequence ($;$)

For $E_b = E_{a_1} ; E_{a_2} ; \cdots ; E_{a_m}$

Rule 4.9: $\exists i \in [1, n], E_a \xrightarrow{F} E_{a_i} \Leftrightarrow E_a \xrightarrow{F} E_b$ with $\text{Correl}(E_a, E_{a_i}) = \text{Correl}(E_a, E_b) \text{Correl}(E_b)$

(3) Conjunction (Any, All)

For $E_b = \text{ANY}(IE_{a_1}, E_{a_2} ; \cdots ; E_{a_m})$

Rule 4.10: $\forall j \in [1, m], (n - I) < m \leq n, E_a \xrightarrow{F} E_{a_j} \Leftrightarrow E_a \xrightarrow{F} E_b$ with $\text{Correl}(E_a, E_{a_j})$

$= \left( \bigcup_{j=1}^{n} \text{Correl}(E_{a_j}, E_b) \right) \text{Correl}(E_b)$
Rule 4.11: \( \forall j \in [1, m], i \leq m \leq n, E_{a_i} \rightarrow^R E_{b_j} \Leftrightarrow E_{a_i} \rightarrow^R E_{b_j} \) with \( \text{CorrelSE}(E_{a_i}, E_{b_j}) \)

\[ (\cup_{j=1}^{m} \text{CorrelS}(E_{a_i}, E_{b_j})) \text{ CorrelS}(E_{a_i}) \]

3) Identify dependency relationships between two composite event types

For two composite event types \( E_a^c = \langle{} OP_a(E_{a_1}, E_{a_2}, \ldots, E_{a_n}), \{\ldots\}, PC_a, \text{CorrelS}_a \rangle \) and \( E_b^c = \langle{} OP_b(E_{b_1}, E_{b_2}, \ldots, E_{b_n}), \{\ldots\}, PC_b, \text{CorrelS}_b \rangle \), if we want to detect the dependency relationships between them, the first step is to verify possible relationships between each member event type pair, in which one comes from \( E_a^c \) and the other comes from \( E_b^c \). Suppose two event types are taken, \( E_{a_i} \) from \( E_a^c \) and \( E_{b_j} \) from \( E_b^c \). We should try to verify whether they have some of four relationship types, that is, \( E_{a_i} \leftarrow E_{b_j}, E_{a_i} \rightarrow E_{b_j}, E_{b_j} \leftarrow E_{a_i} \), or \( E_{b_j} \rightarrow E_{a_i} \). The procedure to verify the existence of a relationship is to do a search following the same type dependency relationships. For example, to detect whether \( E_{a_i} \leftarrow E_{b_j} \) exists or not, we try to search out a set of following relationship links starting from \( E_{a_i} \) and stopping at \( E_{b_j} \). It is possible to find several pieces of same type dependency links, and each link tells us a dependency between \( E_{a_i} \) and \( E_{b_j} \) with the different correlation sets. We should merge these correlation sets.

During the searching process, when an event type appears again in a link, a loop is found and the search stops. It will not appear for a correct process model. But for the loop structure in the process model, this kind of event dependency loop can exist. In this case, the second time appearance of an event type will have a different occurrence index than its first appearance. In the next time search, repetition of this loop can be avoided by trying selecting different consequent event type to continue the search.

After all dependency relationships between each pair of member events are discovered, we can apply Rules 4.2 to 4.11 to determine whether two composite event types have the dependency relationship or not.

5. THE EVENT VIEW SPECIFICATION APPROACH

5.1. The definition of an event view

For an application or a partner, what they care about is a subset of events. One reason is that a human’s attention is a finite resource that must be optimized [35]. If given too little or improperly targeted information, users will act inappropriately or be less effective. With too much information, users must deal with an information overload that adds to their work and masks important information. Another reason is the needs of privacy keeping in inter-organizational collaboration [36]. For example, in B2B, an enterprise may not want the collaborators to know its detailed working process, and only a predefined set of information will be released.

Formally, an event view model is defined as \( EV = \langle{} ETS_{EV}, ER_{EV}, CorrelS_{EV}, filtS_{EV} \rangle \), where \( ETS_{EV} \) is a set of event types, \( ER_{EV} \) is a set of event dependency relationships among event types of \( ETS_{EV} \), \( CorrelS_{EV} \) is a correlation set, and \( filtS_{EV} \) defines a filter.

5.2. Event view specification based on the correlation set and the filter set

To construct an event view automatically, we define \( \langle{} CorrelS_{EV}, filtS_{EV} \rangle \) as the semantic conditions to start with, and then we follow up with a two-step procedure. The first step is to determine event types to be included, and the second step is to obtain related dependency relationships among those event types.

The rules to determine related event types are as follows:

- For a primitive event type, if \( ParaS(E) \subseteq SP_a(CorrelSE) \), then \( E \in EV \).
- For a primitive restricted event type, if \( ParaS(E^r) \subseteq SP_a(CorrelSE) \) and when \( filtS_E = \text{TRUE} \Rightarrow \text{filtSEV} = \text{TRUE} \), then \( E^r \in EV \).
- For a composite event type \( E^c \), if \( CorrelS(E^c) \subseteq SP_a(CorrelSE) \), then \( E^c \in EV \).

For example, if we define \( CorrelSEV = \{ \text{instanceID}, \text{toStatus} \}, \text{filtSEV} = \{ \text{toStatus} = \text{"Completed"} \} \), the events whose occurrences indicate a process step is completed for a process instance are selected out.

For two event types \( E_{a_i} \) and \( E_{b_j} \) that are part of an event view, if they have a restricted dependency relationship, we can check whether this relationship’s filter can be satisfied or not when the
evaluation $\text{filt}_{SEV}$ are evaluated TRUE. If satisfied, the restricted dependency relationship becomes a dependency relationship.

5.3. Event view customization

Although an event view can be constructed automatically, more often than not, an event view will be customized due to some additional reasons. The customization actions include

1. Selecting some event types for publishing.
2. Defining new composite event types for publishing.
3. Selecting some dependency relationships for publishing: some dependency relationships are predefined in the system, and some are automatically discovered.

For a given set of event types, which includes primitive event types and also composite event types, we could apply the rules defined previously to detect the dependencies among these events. Unfortunately, it is possible to introduce the redundant dependencies relationships because these type of relationships are transitive. However, to remove the redundant dependencies among these event types is a transitive reduction problem, which can be computed using its transitive closure.

In some applications, certain event types and dependency relationships are more important or even necessary for business collaboration. This can only be decided by a system analyst. If some common business rules are modeled in a domain, they will likely help us to define a better event view. Because of the limited scope of this paper, we will not go further on this topic here.

We can define different event views for different partners or outside applications. They can access these event views and then subscribe some event types so that they will be notified when corresponding event occurs. Because how to support event dissemination through a pub/sub network has been widely discussed [36], we will not discuss this issue either.

6. THE EVENT VIEW SPECIFICATION FOR BPEL PROCESS

In this section, we will define the patterns to generate the primitive event types and their dependency relationships in terms of the process structure of a BPEL file, which is widely accepted as a process definition language, especially for service-oriented process [15–24].

6.1. An event model for the BPEL process

A service process is composed of a set of ordered activities. When a service process is running, activities are started and then finished according to predefined orders. At the same time, data are transferred from one activity to another. Obviously, all types of state change happening during the process execution can be described as events.

1. Process lifecycle event types

The process lifecycle event can be defined as $E_P = \langle \text{ProcessLifecycleEvent}, \{\text{ProcessID, InstanceID, fromProcessStatus, toProcessStatus}\} \rangle$. Process lifecycle event types can be further divided into the following restricted subtypes:

$$E_{PI} = \langle \text{ProcessInitiated}, \text{ProcessLifecycleEvent}, \{\ldots\} \rangle,$$
$$E_{PR} = \langle \text{ProcessRunning}, \text{ProcessLifecycleEvent}, \{\ldots\} \rangle,$$
$$E_{PS} = \langle \text{ProcessSuspended}, \text{ProcessLifecycleEvent}, \{\ldots\} \rangle,$$
$$E_{PC} = \langle \text{ProcessCompleted}, \text{ProcessLifecycleEvent}, \{\ldots\} \rangle,$$
$$E_{PT} = \langle \text{ProcessTerminated}, \text{ProcessLifecycleEvent}, \{\ldots\} \rangle,$$
$$E_{PF} = \langle \text{ProcessFaulted}, \text{ProcessLifecycleEvent}, \{\ldots\} \rangle.$$
(2) Activity lifecycle event types

The execution of an activity can be described as a state change process. An activity lifecycle event type is defined as \( E_{\text{AS}} = \langle \text{"ActivityLifecycleEvent"}, \{\text{ProcessID}, \text{InstanceID}, \text{ActivityID}, \text{fromActivityStatus}, \text{toActivityStatus}\} \rangle \). The activity lifecycle event type can be further divided into following restricted subtypes:

\[
E_{\text{AR}} = \langle \text{"ActivityReady"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{fromActivityStatus} = \text{\"InActive\"}, \text{toActivityStatus} = \text{\"Ready\"}\} \rangle
\]

\[
E_{\text{AE}} = \langle \text{"ActivityExecuting"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{fromActivityStatus} = \text{\"Ready\"}, \text{toActivityStatus} = \text{\"Executing\"}\} \rangle
\]

\[
E_{\text{AE}} = \langle \text{"ActivityExecuted"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{fromActivityStatus} = \text{\"Executing\"}, \text{toActivityStatus} = \text{\"Executed\"}\} \rangle
\]

\[
E_{\text{AW}} = \langle \text{"ActivityWaiting"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{fromActivityStatus} = \text{\"Executed\"}, \text{toActivityStatus} = \text{\"Waiting\"}\} \rangle
\]

\[
E_{\text{AC}} = \langle \text{"ActivityCompleted"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{fromActivityStatus} = \text{\"Executed OR Waiting\"}, \text{toActivityStatus} = \text{\"Completed\"}\} \rangle
\]

\[
E_{\text{AF}} = \langle \text{"ActivityCompleted"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{toActivityStatus} = \text{\"Terminated\"}\} \rangle
\]

\[
E_{\text{AF}} = \langle \text{"ActivityFaulted"}, \text{"ActivityLifecycleEvent"}, \{\ldots\}, \{\text{toActivityStatus} = \text{\"Faulted\"}\} \rangle
\]

(3) Other event types

There may be other event types defined for a service process model, which are typically data object and temporal information related event types. For example, in BPEL4WS, there are other event types including Expression Evaluated Event: it is triggered when an expression is evaluated to be TRUE or FALSE:

\[
E_{\text{EE}} = \langle \text{"ExpressionEvaluated"}, \{\text{ProcessID}, \text{InstanceID}, \text{conditionExpression}\} \rangle
\]

Message Event: it is triggered by the arrival of an external message:

\[
E_{\text{ME}} = \langle \text{"MessageArrived"}, \{\text{ProcessID}, \text{InstanceID}, \text{message}\} \rangle
\]

Alarm Event: it is triggered by an alarm that goes off after a user-specified period is over:

\[
E_{\text{AE}} = \langle \text{"TimeOut"}, \{\text{ProcessID}, \text{InstanceID}, \text{TemporalExpression}\} \rangle
\]

6.2. Event dependency derivation based on the service process specification

Because a BPEL process model defines the orders that activity execution must follow, it also defines the dependency relationships among event types. Therefore, the primitive event types and their dependency relationships can be derived from the process model. We will discuss this issue in this section.

BPEL supports primitive and structure activities. Primitive activities represent basic constructs and are used for common tasks. A structured activity can contain one or more activities. The first activity in BPEL is a receive activity.

For a process or an activity, it can be terminated for a special purpose or be terminated by a fault from any status. These events can be regarded as deviations from the normal behaviors, which cannot be predicted or completely avoided. We will not take these two event types into account when we discuss how to derive event dependencies in terms of the process model.

Because an event view is defined for the a process model, its event types all include the filter relating to ProcessID. In the following discussion, it is omitted. Because correlation sets for most event type dependencies are obvious, to keep brevity, they will not be listed unless a dependency relationship needs a special correlation set in the following discussion.

To keep brevity, following representations are used:

\[
E_{\text{ARA}} = \langle \text{"Activity A Ready"}, \text{"ActivityReady"}, \{\ldots\}, \{\text{ActivityID} = \text{"A"}\} \rangle
\]

\[
E_{\text{EAC}} = \langle \text{"Activity A Completed"}, \text{"ActivityCompleted"}, \{\ldots\}, \{\text{ActivityID} = \text{"A"}\} \rangle
\]

\[
E_{\text{EE} C_1} = \langle \text{"C_1Evaluated"}, \text{"Expression Evaluated"}, \{\ldots\}, \{\text{conditionExpression} = C_1\} \rangle
\]

\[
E_{\text{MEMM}_1} = \langle \text{"M_1Arrived"}, \text{"MessageArrived"}, \{\ldots\}, \{\text{message} = M_1\} \rangle
\]

\[
E_{\text{ATET}} = \langle \text{"Time T_1Arrived"}, \text{"TimeArrived"}, \{\ldots\}, \{\text{TemporalExpression} = T_1\} \rangle
\]
1) Dependencies among process lifecycle event types
Suppose the first activity in the process model is A, the last activity in the process model is B:
\[ E_{RC} \subseteq \text{Ready}("A") \land \text{Completed}("B") \]
\[ E_{PC} \subseteq \text{Completed}("B") \]

2) Dependencies among a primitive activity’s lifecycle event types
For a primitive activity A, the dependency relationships among its lifecycle events are
\[ \text{Ready}("A") \subseteq \text{Executing}("A") \]
\[ \text{Executing}("A") \subseteq \text{Completed}("A") \]

3) Event dependencies for structured activities
A structured activity can be regarded as a black box so that it has the same behavior with the basic one.
When its structure is considered, its inherent events and their relationship can be derived.
Suppose the structure activity is A and it contains a set of ordered activities \( a_1, a_2, \ldots, a_n \). To keep brevity, in following discussion, the dependencies among the event types of each primitive activity will not be listed.

1. \(<\text{sequence}>\)
A \(<\text{sequence}>\) activity contains one or more activities that are performed sequentially. Suppose \( a_1 \) is the first activity and \( a_n \) is the last one, then the event dependency is
\[ \text{Ready}("A") \subseteq \text{Ready}("a_1") \land \text{Completed}("a_1") \subseteq \text{Completed}("a_2") \land \ldots \land \text{Completed}("a_{n-1}") \subseteq \text{Completed}("a_n") \]

2. \(<\text{if}>\)
An \(<\text{if}>\) activity provides conditional behaviors. The activity consists of an ordered list of one or more conditional branches defined by the \(<\text{if}>\) and optional \(<\text{else}>\) elements, followed by an optional \(<\text{else}>\) element. Suppose \(<\text{if}>\) condition expression \( c_i \) is corresponding to activity \( a_i \) and \( a_{i+1} \) is \(<\text{else}>\) activity.
\[ \text{Ready}("A") \subseteq \text{Evaluated}(C_i) \]
\[ \text{Evaluated}("C_i") \subseteq \text{Ready}("a_1") \land \text{Evaluated}("C_i") \equiv \text{Ready}("a_1") \land \text{Completed}("a_1") \equiv \text{Completed}("a_2") \]
\[ \text{Evaluated}("C_i") \subseteq \text{Ready}("a_2") \land \text{Evaluated}("C_i") \equiv \text{Ready}("a_2") \land \text{Completed}("a_2") \equiv \text{Completed}("a_3") \]
\[ \ldots \]
\[ \text{Evaluated}(C_n) \subseteq \text{Ready}("a_{n+1}") \land \text{Evaluated}(C_n) \equiv \text{Ready}("a_{n+1}") \land \text{Completed}("a_{n+1}") \equiv \text{Completed}("A") \]

3. \(<\text{while}>\)
The \(<\text{while}>\) activity supports repeated performances of an activity in a structured loop. The loop is performed until the specified while condition no longer holds true. Suppose it includes an activity \( a_1 \) inside and the condition is \( C_0 \).
\[ \text{Ready}("A") \subseteq \text{Evaluated}(C_0) \]
\[ \text{Evaluated}(C_0) \subseteq \text{Ready}("a_1") \land \text{Evaluated}(C_0) \equiv \text{Ready}("a_1") \land \text{Completed}("a_1") \equiv \text{Completed}("A") \land \text{Evaluated}(C_0) \equiv \text{Completed}("A") \]

4. \(<\text{repeatUntil}>\)
The \(<\text{repeatUntil}>\) activity provides for repeated execution of a contained activity. The contained activity is executed until the given Boolean condition becomes true. The condition is tested after each execution of the body of the loop. Suppose it includes an activity \( a_1 \) inside and the condition is \( C_0 \).
\[ \text{Ready}("A") \subseteq \text{Ready}("a_1") \land \text{Evaluated}(C_0) \equiv \text{Ready}("a_1") \land \text{Evaluated}(C_0) \equiv \text{Completed}("A") \land \text{Evaluated}(C_0) \equiv \text{Completed}("A") \]

(5) <pick>
A <pick> activity captures race conditions based on timing or external triggers. If more than one of the events occurs, the selection of the activity to perform depends on which event occurred first. Suppose the message arrived event MessageArrived(M) is corresponding to activity \(a_i\) and a time alarm event AlarmHappened(Tj) is corresponding to activity \(a_j\).

\[ \text{Ready}("A") \rightarrow \text{MessageArrived}(M) \]
\[ \text{MessageArrived}(M) \rightarrow \text{Ready}("a_i") \]
\[ \text{Ready}("A") \rightarrow \text{Alarm}(T_j) \]
\[ \text{Alarm}(T_j) \rightarrow \text{Ready}("a_j") \]
\[ \text{ANY}(1, \text{Completed}("a_1"), \text{Completed}("a_2"), \ldots, \text{EndOf}("a_n")) \rightarrow \text{Completed}("A") \]

(6) <forEach>
The <forEach> activity will execute its contained <scope> activity exactly \(n + 1\) times where \(n\) equals the <finalCounterValue> minus the <startCounterValue>. If the value of the parallel attribute is no then the activity is a serial <forEach>. The enclosed <scope> activity MAY be executed \(n + 1\) times, each instance starting only after the previous repetition is complete. If the value of the parallel attribute is yes then the activity is a parallel <forEach>. The enclosed <scope> activity MUST be concurrently executed \(n + 1\) times. The <forEach> activity without a <completionCondition> completes when all of its child <scope>’s have completed. The <completionCondition> element is optionally specified to prevent some of the children from executing (in the serial case), or to force early termination of some of the children (in the parallel case).

Suppose a <forEach> activity is \(A\) and activity \(a_i\) \((i = 1, n + 1)\) is corresponding to the value of counter variable startCounterValue + \(i - 1\).

When \(\text{parallel} = \text{FALSE}:\)
\[ \text{Ready}("A") \rightarrow \text{Ready}("a_i") \]
\[ \text{Completed}("a_i") \rightarrow \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = 1\} \]
\[ \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{F.CompletionCondition}\rightarrow \text{Ready}("a_2")\{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = 1\} \]
\[ \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{Ready}("a_2")\{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = 1\} \]
\[ \ldots \]
\[ \text{Completed}("a_{i-1}") \rightarrow \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = i - 1\} \]
\[ \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{F.CompletionCondition}\rightarrow \text{Ready}("a_i")\{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = i - 1\} \]
\[ \text{Completed}("a_i") \rightarrow \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = i - 1\} \]
\[ \ldots \]
\[ \text{Completed}("a_n") \rightarrow \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = n\} \]
\[ \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{F.CompletionCondition}\rightarrow \text{Ready}("a_{n+1}")\{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = n\} \]
\[ \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{Ready}("a_{n+1}")\{\text{Evaluated}(\text{CompletionCondition}).\text{OI} = n\} \]
\[ \text{Completed}("a_{n+1}") \rightarrow \text{Completed}(A) \rightarrow \text{Evaluated}(\text{CompletionCondition}) \rightarrow \{\text{F.CompletionCondition}\rightarrow \text{Completed}(A)\{\text{Evaluated}(\text{CompletionCondition}).\text{OI} \leq n\} \]
\[ \text{Completed}("a_1") \rightarrow \text{Completed}(A) \]

When \(\text{parallel} = \text{TRUE}:\)
\[ \text{Ready}("A") \rightarrow \text{Ready}("a_i") \ (i = 1, 2, \ldots, n + 1) \]
\[ \text{(Completed("a_1") \lor \text{Terminated("a_1")}) \land \ldots \land (\text{Completed("a_{n+1}")} \lor \text{Terminated("a_{n+1}")}) \rightarrow \text{Completed("A")} \]

(7) <scope>
A <scope> is a special type of structured activity. It is used for grouping activities into blocks. It has a primary activity (i.e. main activity) and provides event handlers, fault handlers, termination handlers and also a compensation handler. Suppose a scope activity is \(A\) and the primary activity is \(a\), we have following rules:
Suppose an activity $A$ and an activity $a_i$ is defined to react to it $(i = 1, 2, \ldots m)$:

$$\text{Completed}(A) \xrightarrow{E} \text{Completed}(a_i)$$

When an event handler is defined, suppose it is $E_{eh}$ and an activity $a_i$ is defined to react to it $(i = 1, 2, \ldots m)$:

$$\text{Completed}(a_i) \xrightarrow{E} \text{Completed}(A)$$

When a termination handler is defined, suppose its activity is $a_j$:

$$\text{Completed}(a_j) \xrightarrow{E} \text{Completed}(A)$$

(9) Compensation handler

When a compensation handler is defined for an activity $A$, suppose it is $E_{ch}$, and an activity $a_i$ is defined to react to it. Suppose the fault is induced by activity $a_{fi}$:

$$\text{Faulted}(a_{fi}) \xrightarrow{E} \text{Completed}(A)$$

(10) <flow>

A <flow> activity provides parallel execution and synchronization of activities.

$$\text{Completed}(A) \xrightarrow{E} \text{Completed}(a_i), \text{Completed}(A) \xrightarrow{E} \text{Completed}(a_2), \ldots, \text{Completed}(A) \xrightarrow{E} \text{Completed}(a_n)$$

The <link> construct is used to express these synchronization dependencies. Declaration of <link>’s are enclosed by a <flow> activity. Each activity has the optional containers <sources> and <targets>, which contain collections of <source> and <target> elements respectively. These elements are used to establish synchronization relationships through a <link>.

Suppose an activity $a$ is the source of link $L_1$ with transition condition $c_1$, then we have following event dependency:

$$\text{Completed}(a) \xrightarrow{E} \text{Evaluated}(c_1)$$

Another activity $b$ with join conditions coming from links with condition $c_1, c_2, \ldots, c_n$, respectively, then we have

$$\text{Evaluated}(c_1) \xrightarrow{E} \text{Evaluated}(b) \ (i = 1, 2, \ldots, n)$$

7. THE CASE STUDY AND THE IMPLEMENTATION ISSUES

7.1. The case study

An order management process is defined in a BPEL file (see Appendix A [24]). As the first step, a hierarchical activity structure can be generated for the process based on the XML structure, which is shown in Figure 6. The graphic view of this process has already shown in Figure 1. This process model can be transformed into a set of events and their dependency relationships as follows:
Filter list:
- $f_1$: goodsAvailable
- $M_1$: order message
- $T_1$: getVariableData('rfQ','payload','rfQ/to')"

The dependency relationships are (The default event dependencies are not listed)

Figure 6. The activity hierarchy structure for an order management process.
The partners should know the progress of the process so that they can define their corresponding behaviors. But for some reasons, the enterprise does not want to or not necessary to provide the whole information to the outside. According to the analysis of business requirements, for example, we want to provide information about

1. A: A RfQ is received;
2. B: A RfQ has been processed and the response has been sent;
3. C: The order is fulfilled successfully;
4. D: The order processing is aborted.

The corresponding event type definitions are

\[ E_A \triangleq \text{Completed}(A_{10}) \]
\[ E_B \triangleq [\text{Completed}(A_{32}) \lor \text{Completed}(A_{35})] \]
\[ E_C \triangleq \text{Completed}(A_{51}) \]
\[ E_D \triangleq [\text{Completed}(A_{42}) \lor \text{Completed}(A_{35}) \lor \text{Completed}(A_{53})] \]

Although the partners can be notified by subscribing above events so that they know the progress, we should provide dependency relationships among these four events for better collaboration. This can be done by applying the mechanism discussed above. We show how to determine the dependency relationship between \( E_A \) and \( E_B \) as an illustration. The following and relying on relationships between \( \text{Completed}(A_{10}) \) and \( \text{Completed}(A_{32}) \), \( \text{Completed}(A_{10}) \) and \( \text{Completed}(A_{32}) \) are verified respectively.
Therefore,

\[
\text{Completed}(A_{10}) \xrightarrow{E_{B}} \text{Completed}(A_{10}) \xrightarrow{E_{B}}
\]

Although the case study presented here is simple, it can be found to define an event view is an easy task because event dependencies can be inferred according to the rules we defined. This will help event view specification become a useful tool to define the collaboration protocols.

7.2. Implementation issues

We have implemented a prototype system, and in this system, an event view can be specified. More specifically, in this system, a new service process model can be designed from scratch and then be implemented into activities with all event dependency relationships being checked and enforced. In addition, a BPEL can also be input to the system by converting the model into a set of event types and event type dependency relationships. Figure 7 shows how to model a process based on event directly.

Figure 7(b) shows the event list of the process model. In addition, complex events can be added to the model. When an event view is specified, event types to be published should first be chosen (Figure 7(c)). The event dependencies will be derived for selected event types, but we can remove some dependencies from the model according to our business strategy (Figure 7(d)).

We also implemented an event broker based on OpenJMS [38] and Esper [27]. The class diagram of the event broker is shown in Figure 8.
Class EventChannelManager is responsible for receiving primitive events from applications and then send it to the event engine. It is implemented in a separate thread by wrapping interfaces of OpenJMS [38]. The complex event detector is based on Esper [27]. It is an open source event engine. This engine is controlled by the object of EventEngineManager. Esper supports the event definition in terms of org.w3c.dom.Node. All complex event types should be declared in a configuration file. When the engine starts, it reads the configuration file, and complex event type definitions are recorded into EventManager. Class ComplexEventListener is responsible for event publishing. It implements the UpdateListerner interface of Esper. When a complex event is detected, this interface is called by the engine. The complex event information is sent to SubscriberBroker object for notifying subscribers. All the subscribers should register their interested events to SubscriberBroker, which record the subscribe information in a database.

8. RELATED WORK

To facilitate collaborations across the organizational boundaries through process technologies is widely applied and extensively researched in recent years [39, 40]. The collaboration patterns were classified into Workflow systems interoperability, Web service choreography and Multi-Phase Process Composition [20]. In the first approach, workflow systems can communicate with each other by exchanging messages to initiate new processes, to change the state of target processes, or to attain the information of process relevant data. For example, in [41], the author presented some forms of workflow interoperability and focused on capacity sharing, chained execution, subcontracting, (extended) case transfer, loosely coupled, and public-to-private architectures. The second approach focuses on implementing a loosely coupled integration for a collaborative process by employing web services. For example, in [42], an architecture based on web service choreography to support collaborative design process is proposed.

The Multi-Phase Process Composition approach is used to implement multiple-phase process design, including both public process and private process. Our approach falls into this category. In [5], an approach is proposed to transform an internal process model into a cooperative process after cooperative activities are identified. Then public processes can be deduced from the cooperative process using the notion of cooperation policy. In [23–43], a novel process view, that is, an abstracted process that is derived from a base process to provide process abstraction, is described for modeling a virtual workflow process. Activities in the base process can be combined together as a virtual activity in an abstracted process model. Besides combination, abbreviation is another way to define the process view. Through abbreviation, some activities are defined as silent activities that are not observable. By renaming activities to silent activities, the desired abstraction can be obtained. Eshuis and Grefen formalized the operations of task aggregation and process customization, and
they also proposed a series of construction rules for validating the structural consistency [44]. Recently, Xiaohui Zhao et al. developed a framework and a series of algorithms to enforce the process abstraction and concretization operations in compliance with structural consistency [45].

All these methods have the assumption that activity is a minimal business transaction unit. Our event view model is more flexible because the execution of an activity is described in terms of several events and events can be composed through different operators. Therefore, we can define very different views for the base process. At the same time, the rules to identify the dependency relationships among events can still make this view consistent with the base model. Although an event view itself is not a process model, we can reconstruct the process model from the event view in terms of the dependency information by incorporating dedicated event types into the view. Therefore, our method is also a more general one.

Event-based process modeling has already widely researched and applied into business activities. Some of the leading products, for example, SAP R/3 (ERP/WFM) and ARIS (BPR), use Event-driven Process Chains (EPCs) to model business processes [46]. In [47], the event calculus as a logic-based methodology for the specification and execution of workflows is proposed. Some researchers try to transform the process model into an event model, for example, in [48], an systematic approach to transform a process model into a set of Event-Condition-Action (ECA) rules are introduced. These products have not provided event view specifications yet, and they can serve as the basis of our model.

Combining SOA and event (CEP or EDA) is becoming increasingly popular. Protocols such as WS Eventing [49] and WS-Notification [50] are introduced. Some service process engine like Apache ODE can generates events to let outside track events of what is exactly happening in the engine [14]. The event model for BPEL was also proposed [51]. There are also some wide area event systems to support event subscribe and notification [52]. They provide possible software architectures to implement our model.

Our model is based on the concept of modeling of dependency relationships among event types. In [29], an approach to check causality between non-atomic events based on causal relationships among atomic events is proposed. Some researchers provide online check method [53]; it helps people capture causality based on event history. But these approaches are not suitable for our semantic event model. Our approach can infer dependency relationships among both primitive and composite event types according to the process definition. This is an extension of traditional causality model.

9. CONCLUSIONS AND FUTURE WORK

To support process collaboration, we need to publish what has been and is happening to outside applications in real time manner. The partners and other applications also need to know dependency relationships among event types so that they can define their responsive behaviors in advance. We provide an event-view-based process collaboration method in this paper. For a process model, we can define different event views for different partners or applications. This paper focuses on how to generate an event view for a process model, especially on how to identify the dependency relationships between complex event types based on dependency relationships among primitive events. The future work will focus on but not limit to

1. Exploring more accurate dependency relationship definition: if we can model dependency with more information, such as cardinality and time delay, the outside partners can know much more about the process.
2. Defining additional rules to help define a better event view for different applications: in this paper, we do not discuss how to define a more optimized event view on-demand. We need set up some detailed matrix to evaluate the quality of an event view. For example, we should be able to judge if the previously defined event view is good enough for a specific collaboration and, if not, how to improve the event view.
ACKNOWLEDGMENTS

The work described in this paper was partially supported by China National Science Foundation (Grant Number 61073021, 61272438) and Research Projects of STCSM (Grant Number 11511500102, 12511502704). It was also partially supported by Morgan Stanley. Morgan Stanley and Shanghai Jiao Tong University Innovation Center of Computing in Financial Services have entered into Collaboration Agreement No.CIP-A20110324-2.
47. Cicekli NK, Cicekli I. Formalizing the specification and execution of workflows using the event calculus. Information Sciences 2006; 176:2227–2267. DOI: 10.1.1.62.6209