An Optimized Derivation of Event Queries to Monitor Choreography Violations

Aymen Baouab, Olivier Perrin, and Claude Godart

Loria - Inria Nancy - Université de Lorraine - UMR 7503,
BP 239, F-54506 Vandoeuvre-les-Nancy Cedex, France
{aymen.baouab,olivier.perrin,claude.godart}@loria.fr

Abstract. The dynamic nature of the cross-organizational business processes poses various challenges to their successful execution. Choreography description languages help to reduce such complexity by providing means for describing complex systems at a higher level. However, this does not necessarily guarantee that erroneous situations cannot occur due to inappropriately specified interactions. Complex event processing can address this concern by analyzing and evaluating message exchange events, to the aim of checking if the actual behavior of the interacting entities effectively adheres to the modeled business constraints. This paper proposes a runtime event-based approach to deal with the problem of monitoring conformance of interaction sequences. Our approach allows for an automatic and optimized generation of rules. After parsing the choreography graph into a hierarchy of *canonical* blocks, tagging each event by its block ascendancy, an optimized set of monitoring queries is generated. We evaluate the concepts based on a scenario showing how much the number of queries can be significantly reduced.

Keywords: web-service choreography, cross-organizational processes, event processing, business activity monitoring.

1 Introduction

The ability of linking cross-organizational business processes is receiving increased attention in an ever more networked economy [1]. Indeed, collaborative computing grows in importance and processes have to deal with complicated transactions that may take days or weeks to complete across wide ranging geographies, time zones, and enterprise boundaries.

Building complex distributed processes, without introducing unintended consequences, represents a real challenge. Choreography description languages help to reduce such complexity by providing means for describing complex systems at a higher level. The birth of a service choreography is often determined by putting together external norms, regulations, policies, best practices, and business goals of each participating organization. All these different requirements have the effect of constraining the possible allowed interactions between a list of partners. However, this does not necessarily guarantee that erroneous situations cannot

C. Liu et al. (Eds.): ICSOC 2012, LNCS 7636, pp. 222–236, 2012.

occur due to inappropriately specified interactions. Indeed, runtime verification must be taken into consideration, to the aim of checking if the actual behavior of the interacting entities effectively adheres to the modeled business constraints.

Run-time monitoring of services composition have been a subject of interest of several research efforts [2–8]. Todays business process monitors mostly use complex event processing (CEP) to track and report the health of a process and its individual instances. During the last ten years, CEP was a growing and active area for business applications. Business activity monitoring (BAM) was one of the most successful areas where CEP has been used. Based on key performance indicators (KPIs), BAM technology enables continuous, near real-time event-based monitoring of business processes. Most commercial BPM software products (e.g. Oracle BAM, Nimbus, Tibco and IBM Tivoli) include BAM dashboard facilities for monitoring, reporting violations of service level agreements (SLAs), and displaying the results as graphical meters. However, such products are typically limited to internal processes that are under control, i.e., intra-organizational setting.

Providing an easy, real-time way to monitor cross-organizational processes, i.e., when each step is executed by a different company in a collaborative network, represents a complicated task. This is due to the fact that monitors have to deal with huge volumes of unstructured data coming from different sources. Moreover, errors may propagate and failures can cascade across partners. By managing aggregations of various alerts, CEP might give business administrators a better visibility, provide accurate information about the status and results of the monitored processes, and help to automatically respond immediately when problems occur.

In this paper, we address the problem of monitoring conformance of interaction sequences with normative cross-organizational process models. We have chosen to take the choreography description as a basis for the generation of rules. We first define a notification event structure by specifying which data it should contain. After parsing the choreography graph into a hierarchy of *canonical* blocks, tagging each event by its block ascendancy, an optimized set of monitoring queries is generated. Derived queries can be directly used in a complex event processing environment. Further, we evaluate the concepts based on a scenario showing how much the number of queries can be significantly reduced.

The rest of the paper is organized as follows. *Section 2* presents a motivating example. *Section 3* illustrates our approach and *Section 4* presents some evaluation results. *Section 5* outlines some implementation guidelines, *Section 6* discusses related work, and *Section 7* summarizes the contribution and outlines future directions.

2 Scenario and Motivation

To illustrate the concepts of our work, we adopt the scenario of a business-to-business choreography involving a customer (C), a supplier (S) and two shippers ($S1$ and $S2$). In this cross-organizational choreography, the customer first interacts with a supplier by sending a request for a quote (message M1) and receiving an

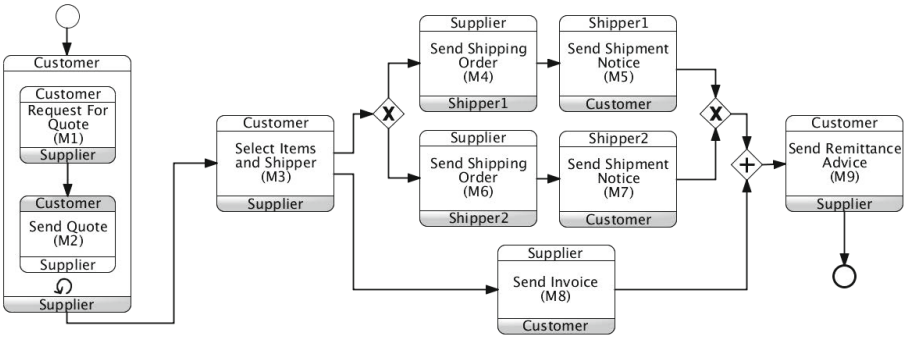


Fig. 1. Cross-organizational choreography example (BPMN 2.0 Diagram)

answer (M2). This step (M1, M2) can be repeated until the customer gets its final decision by selecting items for purchase and the preferred shipper (M3). Then, the supplier transmits a shipping order (M4 or M6) to the selected shipper and sends, in parallel, an invoice to the customer (M8). After finishing the shipment phase, the shipper sends a shipment notice (M5 or M7) to the customer. Finally, the customer proceeds to the payment phase by sending a remittance advice to the supplier (M9) indicating which payables are scheduled for payment. *Figure* 1 shows how to model such a choreography using the BPMN 2.0 notation [9].

During the execution phase, many choreographies shared between different business partners may be instantiated. Indeed, each organization may have multiple external interactions associated with different choreographies instances. Here, there is a need to check the consistency of all incoming and outgoing calls with respect to the current step in each choreography that an organization is participating to. The set of allowed service calls at any given point in time should be dynamic and minimal. Thus, ability to derive instant insights into the cross-organizational interactions is essential. That is, companies should be able to create intelligent agents that monitor their message exchange with the external world in real-time, and automate analysis to detect unusual interaction sequences that fall outside choreography patterns. For instance, a call that is not associated with any current expected step of the instantiated choreographies should be reported to the monitoring applications as a potential violation. Controlling incoming calls at earlier stage may reduce some of common attacks (e.g. DoS attack). Furthermore, it represents a crucial requirement to prevent any malicious peer from exploiting external flow authorizations.

3 Complex Event Queries to Monitor Choreographies

Before we dive headfirst into our approach, we briefly formalize some basic notions . *Section* 3.1 gives background information on Complex Event Processing

(CEP). *Section* 3.2 introduces some formal definitions of a choreography. Afterward, we present the conceptual architecture and we describe the detailed procedure of our approach.

3.1 Complex Event Processing (CEP)

Monitoring business in a highly distributed environment is critical for most enterprises. In the SOA world, CEP can play a significant role through its ability to monitor and detect any deviations from the fixed process models [10]. Indeed, such a technology enables the real time monitoring of multiple streams of event data, allows to analyze them following defined event rules, and permits to react upon threats and opportunities, eventually by creating new derived events, or simply forwarding raw events.

To support real time queries over event data, queries are stored persistently and evaluated by a CEP system as new event data arrives. An *event processor* is an application that analyzes incoming event streams, flags relevant events, discards those that are of no importance, and performs a variety of operations (e.g. reading, creating, filtering, transforming, aggregating) on events objects.

Event processing tools allow users to define patterns of events and monitors are supposed to detect instances of these patterns. Event patterns become more and more sophisticated and queries more complicated. Thus, there is a need to assist the administrator by a semi-automatic generation of event pattern rules from any process model. To identify non-trivial patterns in event sequences and to trigger appropriate actions, CEP techniques to encode the control flow of a process model as a set of event queries have been introduced [11, 12]. Following this strategy, violations can be detected when defined anti-patterns are matched by the CEP engine.

3.2 Formal Foundation (Choreography)

A choreography defines re-usable common rules that govern the ordering of exchanged messages, and the provisioning patterns of collaborative behavior, as agreed upon between two or more interacting participants [13]. In this paper, we perceive a choreography as a description of admissible sequences of send and receive messages between collaborating parties. Our approach focuses on the global behavior of the choreography. Only ordering structures (sequence, parallel, exclusiveness, and iteration) and interaction activities (message exchange activities) are considered. In other words, it is outside the scope of this paper to specify internal activities (e.g. assign activities, silent activities) since they do not generate message exchanges. For the sake of simplicity, we also omit assignment of global variables. We use "participant" and "partner" interchangeably.

We formalize the semantics of a choreography as follows.

Definition 1 (Choreography). *Formally, a choreography* $\mathcal{C}$ *is a tuple* $(\mathcal{P}, \mathcal{I}, \mathcal{O})$ *where*

- $\mathcal{P}$ *is a finite set of participants,*
- $\mathcal{I}$ *is a finite set of interactions,*

– *$\mathcal{O}$ is a finite set of ordering structures defining constraints on the sequencing of interactions.*

An interaction is the basic building block of a choreography, which results in an exchange of information between parties. Every interaction $I \in \mathcal{I}$ corresponds to a certain type of message (e.g. XML Schema), and is associated with a direction of communication, i.e., a source and a destination of the exchanged message. Let $\mathcal{M}_\mathcal{T}$ be the set of message types. Formally, an interaction is defined as follows.

Definition 2 (Interaction). *An interaction $I \in \mathcal{I}$ is a tuple (Iid,s,d,m_t) where Iid is a unique identifier of the interaction, $s,d \in \mathcal{P}$ are the source and the destination of the message, and $m_t \in \mathcal{M}_\mathcal{T}$ is the type of the message.*

The sequencing of interactions is typically captured by four major types of ordering structures:

Sequence. The sequence ordering structure restricts the series of enclosed interactions to be enabled sequentially, in the same order that they are defined.
Exclusiveness. The exclusiveness ordering structure enables specifying that only one of two or more interactions should be performed.
Parallel. The parallel ordering structure contains one or more interactions that are enabled concurrently.
Iteration. An iteration (loop) structure describes the conditional and repeated execution of one or more interactions.

3.3 Approach Overview

Our approach relies only on choreography state changes, i.e., when a global message is sent or received, monitoring only the interactions between the peers in an unobtrusive way, i.e., the exchanged messages are not altered by the monitors and the peers are not aware of the monitors.

Figure 2 provides an overview of the approach. We assume that events that reflect occurrences of message exchanges are provided by one or multiple external

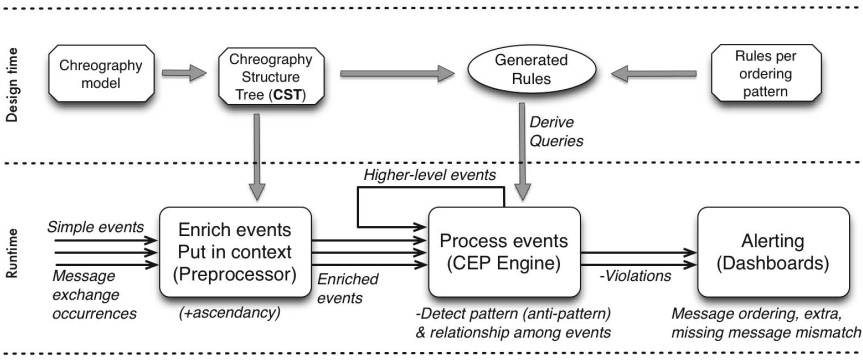


Fig. 2. Conceptual Architecture

components, i.e., event producers. Before being forwarded to a CEP engine, these basic events are enriched by their ascendancy nodes in the structure tree that is derived from the choreography model. Afterwards, the event processor executes predefined complex event queries over the enriched event stream in order to target behavioral deviations during the execution of each choreography. These queries are derived, during design time, from a pre-generated set of rules. Detected violations are sent to be shown in dashboards.

3.4 Basic-Level Events Generation

Most agree, an event is just a fact of something occurring. In the context of a choreography of services, events occur when messages are sent and received. In our case, for each exchanged message, a new notification event is generated. In other words, notification events are generated as transitions occur in the choreography interaction graph. Each notification event is correlated to a choreography message and is generated in order to inform about the occurrence of that message. We define a notification event as follows.

Definition 3 (Notification Event). *A notification $n \in \mathcal{N}$ is a tuple*

$$event = (Eid, Cid, Iid, TS)$$

where Eid is a unique identifier of the event, Cid is a unique identifier for the choreography instance (used for correlation), $Iid \in \mathcal{I}$ is the identifier of the interaction associated to the observed choreography message, and TS is the timestamp of generation).

The field *Cid* is required to correlate events to the different choreography instances. During the execution phase, we may have several instances of different defined choreography models. Obviously, we need to correlate notification events belonging to the same instance. The most common solution [14] to deal with this issue is to define two identifiers that have to be contained in each message (e.g. included in the SOAP header): The choreography ID (a unique identifier for each choreography model) and the choreography instance ID (a unique identifier for each choreography instance). An additional component along the boundary of each participant may be adapted to include and read the identifier whenever a choreography message is exchanged.

The field TS represents the time at which the choreography message is recorded by the event producer. Timestamping events allows for a local ordering through a sequential numbering which is required to analyze the acquired monitoring data.

In this paper, we assume the asynchronous communications among the partners to have an exactly-once in-order reception, i.e., exchanged messages are received exactly once and in the same order in which they are sent. Although this assumption seems to be strong, it is feasible through the adoption of reliable messaging standards (e.g. *WS-ReliableMessaging* [15]).

3.5 Causal Behavioral Profiles

To pinpoint conformance violations during runtime, event rules need to be generated from the fixed interaction-ordering constraints. Following the concept of causal behavioral profiles, [11] proposes to generate a rule between each couple of interaction. As introduced in [16], a causal behavioral profile provides an abstraction of the behavior defined by a process model. It captures behavioral characteristics by relations on the level of activity pairs. That is, a relation is fixed between each couple of activities indicating whether they should be in strict order, in interleaving order, or exclusive to each other. In a choreography, an interaction can be seen as the basic activity. Thus, the same type of relation might be used. For instance, in the model presented in *figure* 1, interactions (M_1) and (M_2) are in strict order. Interactions (M_4) and (M_7) are exclusive to each other. However, interactions (M_5) and (M_8) are in interleaving order. Interleaving order can be seen as the absence of an ordering constraint. Therefore, this relation is not considered when monitoring choreography execution.

Following this approach, in a given choreography with n interactions, the number of rules is equal to n^2. This may overhead the the number of generated queries by extra overlapping ones. For instance, if we have two constraints stating that M_1 should occur before M_2 and M_2 before M_3, then there is no need to fix an additional constraint stating that M_1 should occur before M_3, because this can be deduced automatically. This can be justified by the fact that ordering constraints are transitive. Moreover, when an interaction is performed at an unexpected stage, generated queries may result in multiple redundant alerts. Then, additional queries have to be added in order to identify the root cause for the set of violations as it is done in [11].

Instead of fixing a constraint between each couple of interaction, our approach consists on fixing constraints only between neighbor interactions. To do that, a structural fragmentation of the choreography model is needed.

3.6 Choreography Structure Tree

Until now, we have defined what constitutes a basic level event. In order to reduce the number of constraints, and thus the number of event queries, we provide a decomposition that is inspired from the refined program structure tree (R-PST) of business processes defined in [17] which is a technique for parsing and discovering the structure of a workflow graph. R-PST is proposed as a hierarchical decomposition of a process model into single-entry / single-exit (SESE) *canonical* blocks. A block F is *non-canonical* if there are blocks X,Y,Z such that X and Y are in sequence, $F = X \cup Y$, and F and Z are in sequence; otherwise F is said to be *canonical* [18]. It has been proved that such a SESE decomposition is unique, modular, and can be computed in linear time [17]. In fact, derived blocks never overlap, i.e., given two blocks either one block is completely contained in another block or these blocks are disjoint.

Following this approach, we parse the choreography graph into a hierarchy of SESE *canonical* blocks. *Figure* 3 illustrates *canonical* blocks of our motivating

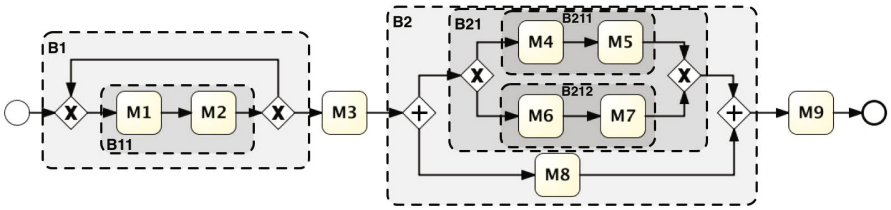
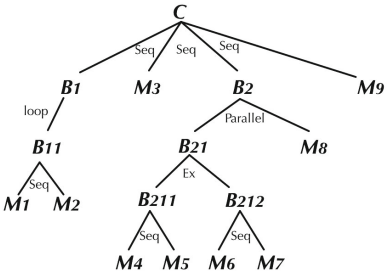


Fig. 3. Fragment decomposition

example. The result of such a decomposition can be represented as a tree that we name Choreography Structure Tree (CST). The largest block that contains the whole graph is the root block of the generated tree. The child blocks of a sequence are ordered left to right from the entry to the exit. *Figure* 4(a) shows the generated CST of our motivation example. We concretize the internal tree nodes by annotating them with the type of ordering pattern, i.e., sequence, parallel, exclusiveness, loop, relating direct descendants. As such, we explicitly establish the structural relation between them.



(a) Resulting CST

$M_1 \rightarrow generate(E_{id}, C_{id}, M_1, < B_{11}, B_1 >, TS)$
$M_2 \rightarrow generate(E_{id}, C_{id}, M_2, < B_{11}, B_1 >, TS)$
$M_3 \rightarrow generate(E_{id}, C_{id}, M_3, <>, TS)$
$M_4 \rightarrow generate(E_{id}, C_{id}, M_4, < B_{211}, B_{21}, B_2 >, TS)$
$M_5 \rightarrow generate(E_{id}, C_{id}, M_5, < B_{211}, B_{21}, B_2 >, TS)$
$M_6 \rightarrow generate(E_{id}, C_{id}, M_6, < B_{212}, B_{21}, B_2 >, TS)$
$M_7 \rightarrow generate(E_{id}, C_{id}, M_7, < B_{212}, B_{21}, B_2 >, TS)$
$M_8 \rightarrow generate(E_{id}, C_{id}, M_8, < B_2 >, TS)$
$M_9 \rightarrow generate(E_{id}, C_{id}, M_9, <>, TS)$

(b) Enriching/Tagging events

Fig. 4. Enriching basic level events

3.7 Enriching Events

After generating the CST, we propose to enrich each basic level event by adding a new field called *ascendancy* containing the list of all superior blocks of the observed message. For instance, the event related to the message (M4) in our motivating example (Figure 3) is tagged by the sequence $< B_{211}, B_{21}, B_2 >$ as it belongs, respectively, to the blocks B_{211}, B_{21} and B_2. This is a kind of tagging each incoming event in order to put it in context, i.e., its supposed location in the CST. After the enrichment step, each basic-level event is transformed into the following structure:

$$event = (Eid, Cid, \text{Iid}, < ascendancy >, TS)$$

This step might be performed by a preprocessor component handling basic event filtering and enrichment. Upon reception of each new basic event, the preprocessor fetches in the CST the ascendancy of the related interaction, and includes

the result as a list of blocks identifiers. *Figure* 4(b) exemplifies how to generate enriched events from the CST and shows the newly enriched events of our motivating example. The produced stream of enriched events serves as input to the main event processor.

3.8 Rules and Higher-Level Events Generation

After enriching events by their superior blocks, rules can be applied to fix block ordering. By doing so, the number of rules decreases exponentially from a level to its higher. Here, we need a generation of higher level events at the end of execution of each block. These newly generated events, indicates the block termination – we note $End(B)$. In the CEP world, a high level event is an event that is an abstraction of other events called its members. In our case, events members are those related to interactions contained in the same block.

To specify these rules, two types of constraints are defined:

- The strict order constraint, denoted by the function $Seq(B_1, B_2)$, holds for two messages M_1 and M_2, respectively tagged with B_1 and B_2, if M_2 never occurs before $End(B_1)$ and M_1 never occurs after M_2.
- The exclusiveness constraint, denoted by the function $Ex(B_1, B_2)$, holds for two messages M_1 and M_2, respectively tagged with B_1 and B_2, if they never occur together within the same choreography instance.

Depending on the ordering patterns, i.e., sequence, parallel, exclusiveness, and iteration, rules are automatically defined to fix when to generate block termination events. *Figure* 5 illustrates the rules for each type of pattern.

Sequence Block. When we have a sequential enactment of n interaction blocks $B_1, ..., B_n$, we can simply enforce the order between each two consecutive blocks, i.e., $Seq(B_i, B_{i+1})$, $i \in \{1..n-1\}$. The completion of the final block in the list induces the generation of the whole block termination event, i.e., $End(B_n) => Generate(End(B))$.

Parallel Block. In case of parallel enactment of n interactions blocks $B_1, ..., B_n$, a violation can only be detected by the absence of one of the internal blocks. Thus, such a violation materializes only at the completion of the whole block B. The completion event is generated only after the termination of all child blocks, i.e, $End(B_1)\&...\&End(B_n) => Generate(End(B))$.

Exclusiveness Block. This choreography construct models the conditional choice. Here, one enactment of only one of the branches is allowed. The decision about which of the branches is enacted is taken internally by one of the participants. In case of n branches of interactions blocks $B_1, ..., B_n$, exclusiveness constraint between each possible couple is generated, i.e., $Ex(B_i, B_j)$, $i \neq j$, $i, j \in \{1..n\}$. The completion event is generated after the termination of one of child blocks, i.e, $End(B_1)$ *or...or* $End(B_n) => Generate(End(B))$.

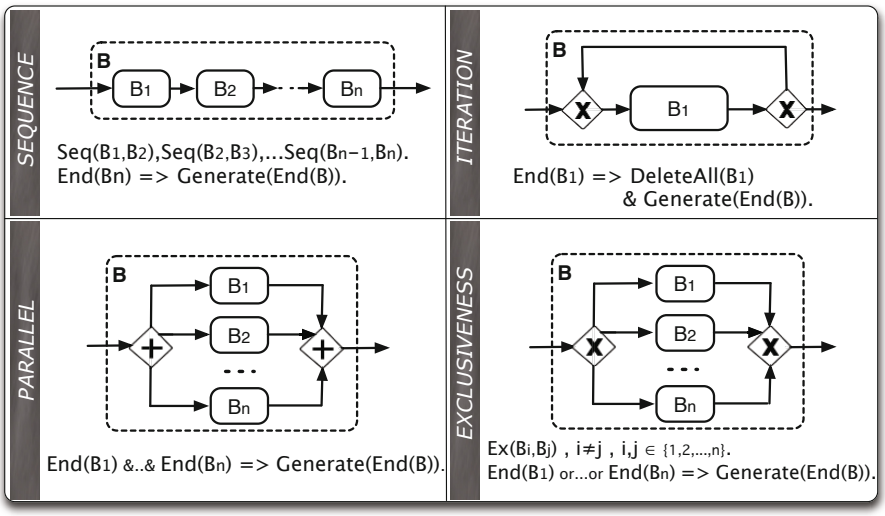


Fig. 5. Rules for each pattern

Iteration Block. In a choreography, an iteration (also called *loop*) activity B specifies the repeated enactment of a branch B_1, which is said to be the body of the iteration. To allow for the repetition of the body's interactions without raising other violations, we need to erase a part of the event history at the end of each iteration and generate the whole block termination event in order to allow the following block to be executed, i.e., $End(B_1)$ => $DeleteAll(B_1)$ & $Generate(End(B))$. Here, $DeleteAll(B_1)$ deletes from the history all events containing B_1 in their *ascendancy* field.

Following this four basic rules and a level per level exploring of the CST, specific rule instances can be automatically generated after the definition of any choreography model. For instance, *table* 1 shows the generated rules from the choreography model presented in *figure* 3. We use the character ';' as a separator between them. The level number represents the depth in the CST.

Table 1. Generated Rules

Level	Generated rules
1	$Seq(B_1, M_3)$; $Seq(M_3, B_2)$; $Seq(B_2, M_9)$; $M_9 \Rightarrow generate(End(C))$
2	$End(B_{11}) \Rightarrow deleteAll(B_{11})$ & $generate(End(B_1))$; $End(B_{21})$ & $M_8 \Rightarrow generate(End(B_2))$
3	$Seq(M_1, M_2)$; $M_2 \Rightarrow generate(End(B_{11}))$; $Ex(B_{211}, B_{212})$; $End(B_{211})$ or $End(B_{212}) \Rightarrow generate(End(B_{21}))$
4	$Seq(M_4, M_5)$; $M_5 \Rightarrow generate(End(B_{211}))$; $Seq(M_6, M_7)$; $M_7 \Rightarrow generate(End(B_{212}))$

3.9 Runtime Pattern Matching

In order to detect violations, rules need to be automatically formulated into event processing queries. Indeed, a generated event query has to match a negation of a rule pattern. For instance, the constraint $Seq(B_1, B_2)$ is violated if and only if the event processor matches any event belonging to the block B_2 that is followed by another event belonging to the block B_1 or simply followed by the high level event $End(B_1)$. However, the constraint $Ex(B_1, B_2)$ is violated when it matches two events, respectively tagged with B_1 and B_2, occur together within the same instance.

When executing the event queries, three types of violation can be detected:

Message ordering mismatch. This violation occurs when the order of messages is not in line with the defined behavior. As an example, when considering the model presented in *figure* 3, let $< M_1, M_2, M_4, M_8, M_3, M_5, M_9 >$ be the sequence of recorded events for one choreography instance. Referring to the generated rules in *table* 1, $Seq(M_3, B_2)$ is twice violated because two messages (M_4 and M_8 that are tagged with B_2) have occurred before M_3.

Extra message mismatch. This violation is detected by the presence of an extra message. It can be matched by a joint occurrence of two exclusive messages. For instance, let $< M_1, M_2, M_4, M_8, M_6, M_5, M_9 >$ be the sequence of recorded events. Here, M_6 is an extra message. $Ex(B_{211}, B_{212})$ is twice violated because two messages (M_4 and M_5 that are tagged with B_{211}) have occurred together with M_6 (that is tagged with B_{212}) within the same instance.

Missing message mismatch. This violation is detected by the absence of a message. This can be materialized only at the competition of the smallest block containing the message. In fact, when the *End* event of this block is not generated, the following sequence is violated. For instance, let $< M_1, M_2, M_3, M_8, M_4, M_9 >$ be the sequence of recorded events. As we can see, M_5 is missing after M_4. Here, $End(B_{211})$, and thus $End(B_{21})$ and $End(B_2)$ are not generated. As M_9 occurred before $End(B_2)$, $Seq(B_2, M_9)$ is then violated.

4 Evaluation

As we can see in *table* 1, we have 14 generated rules. To these rules we may add 9 other rules $Ex(M_i, M_i)$, $i \in \{1..9\}$ in order to indicate that each message should occur only once. Note that this does not affect the messages inside the loop block as their events are deleted at the end of each iteration. These additional rules bring the total number out to 23 (instead of 9x9=81 using the classic behavioral profile approach [11]). Clearly, the benefit of our approach depends on the topology of the CST, e.g., the average number of interactions per blocks, the types of ordering patterns.

For instance, *figure* 6 shows a case where two blocks B_1 and B_2, containing respectively N and M messages in sequence, are exclusive to each other.

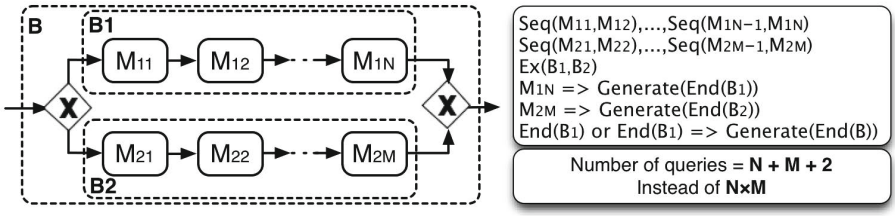


Fig. 6. Special case : Exclusiveness

Here, the total number of generated rules is equal to $N+M+2$ instead of $N\text{x}M$. Clearly, when N and M increase, the difference increases also. For instance, when $N{=}M{=}5$, the number of rules is equal to 12 instead of 25 (48%). However, when $N{=}M{=}10$, the number of rules is equal to 22 instead of 100 (22%). This is one case, among others, that illustrates how our approach can significantly reduce the number of needed queries.

5 Implementation

To execute the queries, we assume the utilization of a CEP engine, coupled with SOAP handlers in order to capture events. First, an input stream and a window to store incoming events on the input stream are created. Queries might be encoded in any dedicated query language that provides pattern definitions. Typical patterns are conjunction, disjunction, negation, and causality. To show the feasibility of our approach, we have chosen to encode the generated queries of our motivating example using the *Esper Processing Language* [19] as it is a commercially proven system and available as open source. This language is a SQL-like language used for querying the inbound event streams. Here, event objects are used instead of specifying tables in which to pull data. The defined queries are registered within the Esper engine in form of event patterns. Then, incoming events are continuously analyzed and processed against these patterns. When matching results are found, defined actions can be undertaken.

Figure 7 shows three queries of our motivating example written in Esper. The first matches if the rule $Seq(M_1, M_2)$ is violated. It detects if a message M_1 is not preceded by a message M_2 that belongs to the same choreography instance ($e2.Cid = e1.Cid$). The second query matches if the rule $Ex(B_{211}, B_{212})$ is violated. It detects any two messages having respectively B_{211} and B_{212} as substrings in their *ascendancy* field and belonging to the same choreography instance. However, the third query matches if the rule $Seq(B_1, M_3)$ is violated. In other words, it detects any message of type M_3 that is not preceded by the generated event $End(B_1)$.

```
 // Matching Seq(M1,M2) violations :
"@Name('Seq M1 M2') select * from pattern "
    + "[ (e2=MsgEvent(e2.iid=2) and not e1=MsgEvent(e1.iid=1))] where e1.cid=e2.cid" ,

// Matching Ex(B211,B212) violations :
"@Name('Ex B211 B212') select * from pattern "
    + "[ e1=MsgEvent(e1.ascendancy like '%B211,%') and "
    + "e2=MsgEvent(e2.ascendancy like '%B212,%')] where e1.cid=e2.cid",

// Matching Seq(B1,M3) violations :
"@Name('Seq B1 M3') select * from pattern "
    + "[ e3=MsgEvent(e3.iid=3) and not b1=MsgEvent(b1.endOf like 'B1')] "
    + "where e3.cid=b1.cid",
/*...*/
```

Fig. 7. Coding event queries using Esper

6 Related Work

Run-time monitoring of services composition have been a subject of interest of several research efforts. In this section we want to outline some of the most relevant contributions with the aim to provide a distinction to our work.

Subramanian *et al.* [20] presented an approach for enhancing BPEL engines by proposing a new dedicated engine called "SelfHealBPEL" that implements additional facilities for runtime detection and handling of failures. Barbon *et al.* [21] proposed an architecture that separates the business logic of a web service from the monitoring functionality and defined a language that allows for specifying statistic and time-related properties. However, their approach focus on single BPEL orchestrations and do not deal with monitoring of choreographies in a cross-organizational setting. Ardissono *et al.* [2] presented a framework for supporting the monitoring of the progress of a choreography in order to ensure the early detection of faults and the notification of the affected participants. The approach consists on a central monitor which is notified by each participant whenever he sends or receives a message.

In case of decentralized processes within the same organization (or within a circle of trust), Chafle *et al.* [7] have modeled a central entity as a status monitor which is implemented as a web service. On each partition, a local monitoring agent captures the local state of the composite service partition and periodically updates the centralized status monitor. The status monitor maintains the status of all the activities of the global composite service. In [8], the authors introduce the concept of monitor-based messenger (MBM), which processes exchanged messages through a runtime monitor. Each local monitor stamps its outgoing messages with the current monitor state to prevent desynchronizations, provide a total ordering of messages, and offer protection against unreliable messaging.

Regarding event-centric perspectives, process monitoring solutions focus on intra-organizational processes and are mostly based on Business Activity Monitoring (BAM) technology [22]. To the best of our knowledge, only two event-centric approaches deal with monitoring cross-organizational choreographies.

The first one [23] uses a common audit format which allows processing and correlating events across different BPEL engines. The second approach [14] introduces complex event specification and uses a choreography instance identifier (ciid) to deal with event correlation (which is not supported in [23]).

In contrast to the previously mentioned works, we rather focus on providing an approach for the automated generation of an optimized set of monitoring queries from any choreography specification. These queries are then directly used in a CEP environment. Weidlich *et al.* [11] proposes a formal technique to derive monitoring event-based queries from a process model. Following the concept of causal behavioral profiles [16], authors propose to generate a rule between each couple of interaction. Then, additional queries have to be added in order to identify the root cause for the set of violations. Instead of fixing a constraint between each couple of interaction, our approach consists on fixing constraints only between neighbor blocks of interactions.

7 Conclusion and Future Work

In this paper, we have proposed an approach for monitoring message exchange deviations in cross-organizational choreographies. Our contribution is a formal technique to generate event-based monitoring queries that match message ordering violations. We have demonstrated that after parsing the choreography graph into a hierarchy of *canonical* blocks, and tagging each event by its block ascendancy, our approach allows to significantly reduce the number of needed queries. Furthermore, we have shown how can these queries be directly used in a CEP environment by providing implementation guidelines.

As future work, we plan to investigate the efficiency of our approach for different types of choreographies. Moreover, we aim to enhance it by providing additional monitoring features to address some quality of service concerns. For instance, it would be interesting to deal with delays in message exchanges. To this end, time constraints violations might be calculated by fixing timeouts and expected time to elapse between messages.

References

1. Grefen, P.: Towards dynamic interorganizational business process management. In: Enabling Technologies: Infrastructure for Collaborative Enterprises (2006)
2. Ardissono, L., Furnari, R., Goy, A., Petrone, G., Segnan, M.: Monitoring choreographed services. In: Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, CISSE 2006, pp. 283–288 (2006)
3. Francalanza, A., Gauci, A., Pace, G.: Runtime monitoring of distributed systems (extended abstract). Technical report, University of Malta, WICT (2010)
4. Moser, O., Rosenberg, F., Dustdar, S.: Event Driven Monitoring for Service Composition Infrastructures. In: Chen, L., Triantafillou, P., Suel, T. (eds.) WISE 2010. LNCS, vol. 6488, pp. 38–51. Springer, Heidelberg (2010)
5. Baouab, A., Fdhila, W., Perrin, O., Godart, C.: Towards decentralized monitoring of supply chains. In: 19th IEEE International Conference on Web Services, ICWS (2012)

6. Baouab, A., Perrin, O., Godart, C.: An event-driven approach for runtime verification of inter-organizational choreographies. In: 2011 IEEE International Conference on Services Computing, SCC (2011)
7. Chafle, G.B., Chandra, S., Mann, V., Nanda, M.G.: Decentralized orchestration of composite web services. In: Proceedings of the 13th International World Wide Web Conference, WWW Alt. 2004. ACM, New York (2004)
8. Halle, S., Villemaire, R.: Flexible and reliable messaging using runtime monitoring. In: 13th Enterprise Distributed Object Computing Conference Workshops, EDOCW 2009 (September 2009)
9. OMG: Business process model and notation (bpmn), version 2.0 (2011)
10. Etzion, O., Niblett, P., Luckham, D.: Event Processing in Action. Manning Pubs. Co Series. Manning Publications (2010)
11. Weidlich, M., Ziekow, H., Mendling, J., Günther, O., Weske, M., Desai, N.: Event-Based Monitoring of Process Execution Violations. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 182–198. Springer, Heidelberg (2011)
12. Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J., Weske, M.: Process compliance analysis based on behavioural profiles. Inf. Syst. 36(7) (November 2011)
13. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web services choreography description language version 1.0. W3C (2005)
14. Wetzstein, B., Karastoyanova, D., Kopp, O., Leymann, F., Zwink, D.: Cross-organizational process monitoring based on service choreographies. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC 2010 (2010)
15. Fremantle, P., Patil, S., Davis, D., Karmarkar, A., Pilz, G., Winkler, S., Yalçinalp, U.: Web Services Reliable Messaging (WS-ReliableMessaging). OASIS (2009)
16. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Efficient Computation of Causal Behavioural Profiles Using Structural Decomposition. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 63–83. Springer, Heidelberg (2010)
17. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified Computation and Generalization of the Refined Process Structure Tree. In: Bravetti, M., Bultan, T. (eds.) WS-FM 2010. LNCS, vol. 6551, pp. 25–41. Springer, Heidelberg (2011)
18. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
19. EsperTech: Esper - Complex Event Processing (2011), `http://esper.codehaus.org`
20. Subramanian, S., Thiran, P., Narendra, N., Mostefaoui, G., Maamar, Z.: On the enhancement of bpel engines for self-healing composite web services. In: International Symposium on Applications and the Internet, SAINT 2008 (2008)
21. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: IEEE International Conference on Web Services (2006)
22. Dahanayake, A., Welke, R.J., Cavalheiro, G.: Improving the understanding of bam technology for real-time decision support. Int. J. Bus. Inf. Syst. 7 (2011)
23. Kikuchi, S., Shimamura, H., Kanna, Y.: Monitoring method of cross-sites' processes executed by multiple ws-bpel processors. In: CEC/EEE 2007 (2007)