An Event-Driven Approach for Runtime Verification of Inter-Organizational Choreographies

Aymen Baouab, Olivier Perrin, Claude Godart

An event-driven approach for runtime verification of inter-organizational choreographies

Aymen Baouab, Olivier Perrin, and Claude Godart
LORIA - INRIA - CNRS
Nancy University - UMR 7503
BP 239, F-54506 Vandoeuvre-les-Nancy Cedex, France
Email: {firstname.lastname}@loria.fr

***Abstract*—In todays economy, collaborative computing grows in importance. Inter-organizational service-based processes are increasingly adopted by different companies when they cannot achieve goals on their own. As a result, conformance problems arise and it must be ensured that the integrity of processes execution remains guaranteed. In this paper, we propose new components, to be deployed along the boundaries of each participating organization, offering external flow control, and notification in case of violation detection, while providing process execution traceability. To achieve our goals, we propose an event-based approach in which inter-organizational exchanges are perceived as events. We define event patterns for filtering the desirable incoming and outgoing messages.**

***Keywords*-Inter-organizational flow; Business process; Choreography; Service notification; event.**

I. Introduction

The ability of linking inter-organizational business processes is receiving increased attention in an ever more networked economy [7]. Nowadays, using electronic data interchange (EDI) for trading relationships among businesses does not meet the demands as EDI mainly enables bi-lateral relationship specific linkages without supporting dynamic collaboration and inter-process interactions. However, many cross-organizations projects have a need for business-process collaboration support.

As opposed to intra-organizational centralized processes, the decentralized setting of those deployed across organizational boundaries raises new control requirements. Without a centralized coordinator that can be a performance bottleneck in some business scenarios, collaborating parties must be able to set up an inter-organizational business process by disclosing to each other only as much as necessary.

In this paper, we consider that the collaborating parties reveal only part of their behavior and processes, keeping internal business activities invisible to other parties. Thus, the global process can be seen as a choreography, which defines re-usable common rules that govern the ordering of exchanged messages, and provisioning patterns of collaborative behavior, as agreed upon between two or more interacting participants [8]. In this context, a first important aspect deals with the conformance. A choreography can sometimes fail as a result of an exceptional circumstance or an error that occurred during its execution. The number of exceptions may increase when the coordination is not well supervised. Indeed, when participants cannot rely on a trusted centralized coordinator to manage basic execution primitives such as message routing, basic security features such as the integrity of the process execution are no longer guaranteed. In fact, additional verifications must be added within each business partner in order to ensure the conformance of the overall sequence of operations with the predefined process execution plan.

A second aspect deals with the way to ensure that a choreography is conform to what is expected, without too much overhead to the in-place architecture. To meet this need, additional components have to be deployed in an agile and flexible way within each organization's boundaries. Hence, there is a need of separating these non-functional aspects from the business-process, following the principle of *separation of concerns* as we proceeded in a previous work [3].

The aim of this paper is to address these issues by exploring inter-organizational business collaborations in a technology-independent way and by providing a new conceptual model. We are focusing, on the one hand, on a distributed, agile and flexible manner to ensure the conformance of the overall sequence of inter-organizational choreography operations with a predefined process execution plan and, on the other hand, on a dynamic and efficient decentralized architecture for errors notifications, while providing process execution traceability.

The approach we introduce in this paper consists in providing new components to be deployed within each participating organization. The purpose of these components is threefold :

1) Message interception : Being deployed along the boundaries of each participating organization, our components have to be able to intercept all exchanged messages with the outside environment. These intercepted messages are then filtered and controlled.

2) Message ordering verification : Ensuring that the sequence of sent and received messages is consistent with the predefined plan. The purpose is to verify whether participating parties have performed their tasks according to the used control flow constructs of the choreography (i.e. the control-flow is proceeded correctly by known participants who behave as expected).
3) Event-driven notification : Selective notification of the involved partner(s) in case of inconsistent flow detection.

To achieve these goals, we propose an event-based approach in which inter-organizational exchanges are perceived as events. Indeed, dealing with issues mentioned above in an event-based way allows for more flexible, scalable and dynamic scenarios running in a highly-reconfigurable infrastructure. Following the principle of Complex Event Processing (CEP) [11], we use event patterns to filter, monitor, and check for the desirable incoming and outgoing calls.

This paper presents a motivating scenario in $Section\ II$, then proposes an architecture and a policy generation mechanism in $Section\ III$, and describes the proposed event-based approach in $Section\ IV$. $Section\ V$ presents some deployment guidelines. $Section\ VI$ describes some related work. Finally, $Section\ VII$ concludes the paper and outlines future directions.

II. Scenario and motivation

This section presents a classical scenario and discusses the motivation behind the proposed research.

A. Motivating example

For explaining the motivation and concepts of our work we have chosen a classical purchase order scenario as illustrated in the diagram shown in *Figure* 1. The diagram shows a choreography between a reseller, a supplier and two shippers (other involved participants such as banks have been omitted for space reasons). The reseller sends a purchase order request with details about the required products and the needed amounts to the supplier (1). The supplier checks the order, then notifies the reseller if and only if there is no more available products ($2c$). Otherwise, the supplier processes the order and selects one out of two shippers to deliver the selected products ($2a$ or $2b$). In this case, the chosen shipper finishes the process by sending the products to the reseller ($3a$ or $3b$).

Without any trusted coordinator to refer to, each business partner participating in the choreography needs to be able to verify, at any given point in time, that all incoming and outgoing messages conform to the expected ones according to the predefined process plan. Thus a control-flow aware policy is needed. For instance, when considering the scenario of *Figure* 1, the supplier has to choose between three service

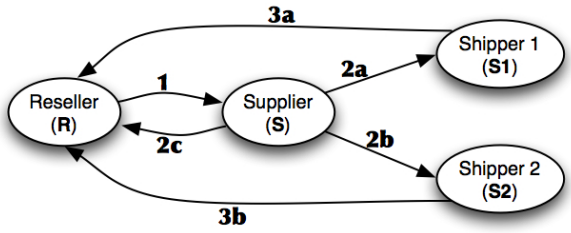

Figure 1. Motivating scenario.

calls (i.e. $2a$, $2b$ or $2c$) depending on its internal subprocess execution (which is not visible to other participants). In return, the reseller will receive one, and only one of the messages ($2c$, $3a$ or $3b$). If it is not the case, the Reseller must detect a non compliant behavior regarding the predefined execution plan and may have to inform other involved partners. Indeed, in such a case, the Reseller may be the only collaborating party that is aware of what happened and able to notify the other parties, what is of first interest in the absence of a central coordinator.

B. Motivation of the approach

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. Thus, 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. Access to resources must be restricted pursuant to the *principle of least privilege* [17]. When data is passed between participants, there must be access restrictions according to the tasks. The set of allowed service calls at any given point in time should be dynamic and minimal. 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) and acts as a defense against malicious attacks that may aim to exploit external flow authorizations. Furthermore, it represents a crucial requirement to prevent any malicious peer from forging a workflow instance.

Our approach aims at providing a simplified mechanism that ensures the conformance of the inter-organizational flow and the correctness of the execution order of the choreography. Before presenting the approach, we model a choreography as a set of interactions and control-flow elements (sequence, parallel, choice, etc.). *Figure* 2.a. shows the global view of the choreography presented in the previous scenario.

In order to be able to control the whole process execution in a decentralized way (i.e. without a coordinator), a global state could be permanently maintained and shared among the participants during the execution phase. In such a case, additional exchange of messages would be required which would complicate our task. To address this issue, we chose to have each party only controlling its own interactions, and thus no current global states are exchanged. Thus, parties that are not involved in the current interactions only approximate the current view. Certainly, this may relax the achievement of our goals but frees us from a synchronization overhead.

To control its own interactions, the global view is no more needed and each party computes its own local view of the choreography. As shown in [9], each participant can do this by simply filtering all interactions that do not have itself as a source or a destination. Afterwards, unnecessary nodes and control-flow elements are removed. *Figures* 2.b, 2.c, 2.d and 2.e show the calculated local view of each organization participating in our scenario.

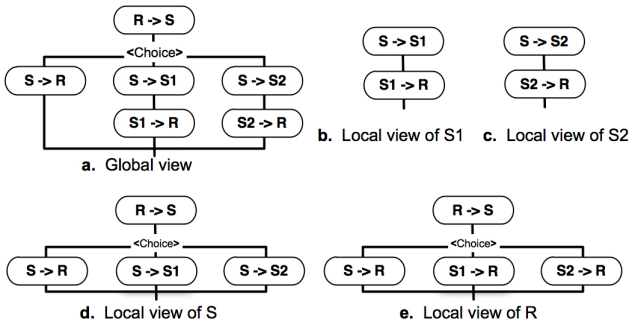

Figure 2. Choreography views.

III. Overview of the Approach

In this section, a policy generation mechanism that have to be executed during the set-up phase of the choreography is presented. Afterward, a short overview of the proposed architecture is provided.

A. External flow policy generation

After the creation of the local view of each choreography, a local policy for external flow control is automatically created within each participant. This policy specifies additional constraints for the received and sent messages based on the control-flow dependencies. These dependencies are generated from the local view of the choreography by a simple calculation of the predecessor(s) of each incoming and outgoing message. Based on the preceding control-flow pattern, this can be achieved automatically by using any predecessor calculator depending on the used choreography language. The policy is defined as follows.

Definition. *An external flow policy (EFP) is 5-tuple of*

$$< id, S/R, M_{type}, Src/Dst, Pred_{cond} >$$

where id is the policy identifier, S/R is the message direction (Receive or Send), M_{type} is the message type (e.g. XML Schema), Src/Dst is the involved partner (as source or destination), and $Pred_{cond}$ is the predecessor condition to be satisfied (i.e. the list of the messages that must be detected before).

This policy allows to accept only the right message from/to the right partner at the right step of the choreography execution. Messages that do not match any entry of the policy table must be pointed out. In order to make it more explicit, *Figure* 3 shows the locally generated policy of each organization involved in the scenario presented in Section II-A. As opposed to dynamic policies such as [9] and [20]

Allowed messages				$Pred_{cond}$ (Id)
Id	S/R	M.type	Src/Dst	
E1	Send	1	S	-
E2	Receive	2c	S	E1
E3	Receive	3a	S1	E1
E4	Receive	3b	S2	E1

a) EFP of the Reseller

Allowed messages				$Pred_{cond}$ (Id)
Id	S/R	M.type	Src/Dst	
E1	Receive	1	R	-
E2	Send	2c	R	E1
E3	Send	2a	S1	E1
E4	Send	2b	S2	E1

b) EFP of the Supplier

Allowed messages				$Pred_{cond}$ (Id)
Id	S/R	M.type	Src/Dst	
E1	Receive	2a	S	-
E2	Send	3a	R	E1

c) EFP of Shipper 1

Allowed messages				$Pred_{cond}$ (Id)
Id	S/R	M.type	Src/Dst	
E1	Receive	2b	S	-
E2	Send	3b	R	E1

d) EFP of Shipper 2

Figure 3. External flow policies

that suffer from scalability problems, since for each process instance a new policy have to be instantiated, our policy is static and, thereby, shared by all the instances. Indeed, one, and only one, policy is generated after the creation of the choreography.

B. Proposed architecture

In order to present the concept in a technology-independent way, we prefer to outline a general architecture rather than describing technical details related to any service

choreography languages, such as WS-CDL[1], BPEL4Chor [5] or Let's Dance [21]. We propose to extend any organization architecture with two components :

The *External Flow Controller (EFC) :* Responsible for intercepting, controlling and monitoring incoming and outgoing messages. Two types of control are depicted: incoming message control in order to restrict the access to internal resources taking into account the context of the call and the control-flow of the involved inter-organizational process, and outgoing message control aiming to prevent from divulging sensitive information in unexpected ways (i.e. without a consistent inter-organizational process context).

The *External Flow Policy (EFP) Repository :* In which external flow policy is generated and stored. To allow for a policy-based approach, it is important to separate this component from the EFC. The EFP Repository can be managed centrally by a governance layer. The policies can be then audited, traced, and historically managed.

From a technical point of view, each component can be implemented as a set of specialized services. *Figure* 4 presents an abstract view of the organization architecture. A further technical description follows.

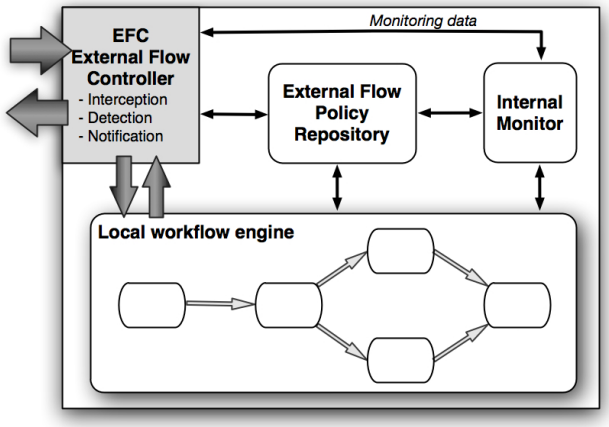

Figure 4. Organization architecture.

IV. Event-driven message verification

Instead of periodically checking (with a request/response) whether a situation have been detected or not, verifications and notifications are performed only when they need to (i.e. when there is a new message or an interesting information to report). To do so, we propose an event-based approach in which each message received or sent can be associated to an event.

First of all, event patterns have to be defined in order to intercept desirable incoming and outgoing calls. Then, these patterns have to be implemented into event processing agents within each organization boundaries. To do so, we use the RAPIDE *Event Pattern Language* [11], a strong-typing declarative computer language for writing patterns of events. It was designed to be commonly used for *Complex Event Processing* (CEP) [10], which is a layer built on top of *Event Driven Architecture* (EDA). Even if there are many others event processing language projects [10] (e.g. STRAW-EPL, StreamSQL, Infospheres [1], AMiT [12]), the choice of this language is motivated by its notational simplicity and efficiency of pattern matching versus power of expression. However, the use of any other *EPL* remains possible. It should be noticed that each *EPL* has a pattern matcher which is a program that processes one or more event executions in real time and picks out all, and only, posets that match a defined pattern. In our case, the pattern matcher is implemented within the EFC.

A. Definition of events

The patterns specify sets of events together with their parameters, timestamps, and eventual causal dependencies. Defining events consists of declaring the types of parameters followed by the events templates. In our approach, we first distinguish three input types of event : *Send* (outgoing call), *Receive* (incoming call), *RecNotf* (incoming notification), and one output type : *Notify* (generated notification). Then we discern five parameters : the choreography instance identifier, the destination or the source partner identifier, the message type (e.g. an *XML schema* identifier), the message identifier and an optional field for eventual error information. *Figure* 5 shows how to define these events and their parameters using the RAPIDE-EPL syntax.

```
typedef Instance ... ; -- the instance Id of the choreography
typedef Org .... ; -- The organization Id
typedef MsgType .... ; -- The type of the message
typedef MsgId .... ;  -- A unique message identifier
typedef Info .... ;  -- i.e. Error description
in action Send(Instance i, Org dest, MsgType mt, MsgId id);
-- Sending a message to the destination dest.
in action Receive(Instance i, Org src, MsgType mt, MsgId id);
-- Receiving a message from  a source src.
in action RecNotf(Instance i, Org src, MsgType mt, MsgId id, Info i);
-- Receiving a notification from  a source src. Here src can be the
-- organization of the agent (i.e. internal notification).
out action Notify(Instance i, Org dest, MsgType mt, MsgId id, Info i);
-- Sending a notification from  to a destination dest. Here dest can be
-- the organization of the agent (i.e. internal notification).
```

Figure 5. Events definition using RAPIDE-EPL.

B. Definition of Event Pattern Rules

An event pattern rule is a reactive rule with two parts: a left side called the *trigger* (which is a RAPIDE-EPL pattern) and a right side called the *body* (which is a list of statements). Each rule has the following syntax :

[1]Web services choreography description language

$$pattern => body(list - of - statements);$$

We propose to set up a trigger for each type of event ($Send, Receive$ and $RecNotify$). In case of each $Send$ or $Receive$, the pattern matcher matches the corresponding entry in the local EFP table and checks whether the predecessor condition is satisfied. If not, a notification event is generated. Our basic rules are shown in *Figure* 6. The use of "?" as a prefix to a pattern parameter helps distinguish the variable parts of a pattern from the constant parts. A *generate* statement in a rule's body creates new events that usually belong to the output action set. In our case, the *Notify()* event, which is fired in case of each policy violation, is the only output event.

```
1  SetofEvents Set; // used for traceability..
2  Org = ... ; // the organization in which the agent is implemented.
3  .......
4  // Functions
5  Boolean   Policy.find(String action, Org o, MsgType mt);
6  // fetches if an associated policy exists..
7  SetofEvents Policy.getdependencies(Instance i, MsgType mt);
8  // returns from the policy the associated predecessor(s)
9  Boolean  Set.Find(SetofEvents list);
10 .......
11 // Event pattern rules for outgoing messages
12 Send(?i, ?dest, ?mt, ?id) =>
13         if Policy.find("Send", ?dest, ?mt)  {
14             if Set.Find(Policy.getdependencies(?i, ?mt))
15                 Set.Store(Send(?i, ?dest, ?mt, ?id)) ;
16             else
17                 generate Notify(?i, Org, ?mt, ?id, "Order exec error..");
18         } else
19             generate Notify(?i, Org, ?mt, ?id, "Undefined Msg type..");
20 .......
21 // Event pattern rules for incoming messages
22 Receive(?i, ?src, ?mt, ?id) =>
23         if Policy.find("Receive", ?src, ?mt)  {
24             if Set.Find(Policy.getdependencies(?i, ?mt))
25                 Set.Store(Receive(?i, ?dest, ?mt, ?id)) ;
26             else
27                 generate Notify(?i, Org, ?mt, ?id, "Order exec error..");
28         } else
29             generate Notify(?i, Org, ?mt, ?id, "Undefined Msg type..");
30 ........
31 // Event pattern rules for received notifications
32 RecNotf(?i, ?src, ?mt, ?id, ?i) =>
33         generate Notify(?i, Org, ?mt, ?id, "Received notification from..");
34 ........
```

Figure 6. Event Pattern Rules definition

The $Policy.find()$ function, defined on line 5, accepts three parameters : the action (which is send or receive), the organization as a source or a destination (depending on the action) and the message type. This boolean function checks whether an action defined by these three parameters exists in the local EFP table. We remind that the latter takes the form shown in *Figure* 3.

The $Policy.getdependencies()$ function, defined on line 7, accepts two parameters : the instance identifier and the message type. This function returns the list of the predecessors related to that message type from the local EFP table.

The $Set.Find()$ function, defined on line 9, accepts a list of events as parameter and returns whether the events of this list exist in Set (i.e. the set of events, defined on line 1, which contains the history of all accepted events and which can be used also for traceability issues).

Finally, the $Set.Store()$ function, stores the event associated to each accepted message in the list of events Set.

After the definition of these functions, event pattern rules are depicted with the implementation of a list of statements for each defined trigger. We distinguish three basic event triggers :

Outgoing calls trigger (line 12): This trigger is matched whenever an outgoing call is detected. For each match, the body (lines 13 to 19) is executed. First, a policy enforcement is fulfilled by checking whether a corresponding entry exists in the local EFP table. Then, the execution order is checked by verifying that all events that should happen before (referring to the corresponding *Predecessor* field in the policy) have been already matched and stored in the event history set.

Incoming calls trigger (line 22): This trigger is matched whenever an outgoing call is detected. For each match, the body (lines 23 to 29) is executed. The body statements are almost the same as those of outgoing calls rules. The unique difference is that the action field is instantiated to *Receive* instead of *Send.*

Received notifications trigger (line 32): This trigger is matched whenever an external notification is received. For each match, the notification is simply forwarded by generating an internal notification.

C. Extensibility : Additional Rules

Flexibility represents an important non functional aspect of our approach. Indeed, it is important that the EFC offers the ability to be changed whenever the security requirements of the organization change evolve. Thus, an interface for rules management is advised in order to permit easy implementation of additional rules. Each additional rule should have a security purpose behind it. Thus, adding new rules fixes additional constraints with the intention of denying some attacks or avoiding conflicts.

Figure 7 shows an example of additional rules. The independence operator, $pattern1$ II $pattern2$, indicates that all the events matching $pattern1$ must be independent of all the events matching $pattern2$.

For instance, $Rule$ 4 (line 37) verifies whether each message identifier is unique. If it is not the case (i.e. when two messages having the same ID are detected), a new notification event is fired.

Furthermore, $Rule$ 5 (line 42) verifies whether messages having the same message type within the same choreography instance (i.e. same instance identifier) are detected. This can

```
36 // Rule 4 : Calls having the same message ID
37 Receive(?i1, ?src1, ?mt1, ?id) II Receive(?i2, ?src2, ?mt2, ?id) =>
38             generate Notify(Org, "Order exec error..");
39 .....
40 // Rule 5 : Calls having the same message type within the
41 //same instance
42 Receive(?i, ?src1, ?mt, ?id1) II Receive(?i, ?src2, ?mt, ?id2) =>
43             generate Notify(Org, "Order exec error..");
44 .....
45 // more additional rules..
```

Figure 7. Additional Rules

be a result of a $Retry$ action when a message is lost and a reliable messaging mechanism is used, or in case of a $rollback$ action when a global exception occurs. Hence, more sophisticated verifications and actions dealing with these issues can be added in the $body$ of the rule.

D. Notifications

We distinguish two types of notifications : internal notifications that are typically generated by the EFC and bound for an internal monitor (i.e. within the same organization) and external ones that are exchanged by EFCs of different organizations (i.e. crossing organization boundaries).

Internal notifications: For each detected noncompliance with the predefined process execution plan, the EFC notifies its own organization by sending an error message and an overview of the involved calls to an internal monitor. As explained previously, the EFC stores information related to all incoming and outgoing calls of its organization. This information can be useful for some internal components of the organization the EFC belongs to.

External notifications: In some case of errors, it deems advisable that the EFC notifies the involved partner(s). This may permit to detect not viable parts of the global business process in a earlier stage. A notification message may at least contain an identification of the message of origin and an error code. Such information will be useful for the involved partner in order to raise and handle an exception in the main choreography. This may save time compared to the traditional way (i.e. when exceptions are managed by the choreography language) especially in the case of internal exceptions that cannot be detected from the outside. In such a case, timeout counters are traditionally used but are not well adapted to long-running collaborations.

In counterpart, received notifications are transformed into internal ones in order to be forwarded inside the organization.

V. Deployment Guidelines

To demonstrate the feasibility of our concept, a preliminary reference architecture is proposed in *Figure* 8. Typically, the EFC may be deployed as a gateway service intercepting all incoming and outgoing messages and notifying when necessary. Allowed incoming messages, after being verified by an authorization system (e.g. PEP/PDP[2] such as defined in XACML[18]), can then be forwarded within the secure zone in which resources have been deployed. In the other direction, outgoing messages are checked for authorization then intercepted by the EFC before being sent outside. In case of complex architectures, it is advisable to

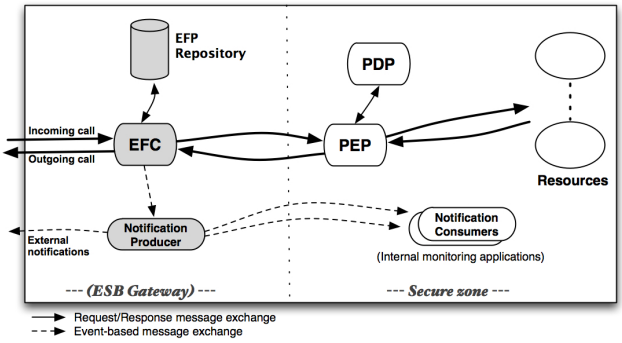

Figure 8. Deployment within an organization

follow the principle of *Separation of Concerns* by deploying each functionality as a separate service (e.g. message intercepter, message analyzer, notifier, etc.) and making them collaborating together. Moreover, the EFP Repository may also be deployed as a set of services offering interfaces to access, monitor and manage generated policies. This strategy may increase the agility of our components allowing for simpler future upgrades.

In order to allow for the possibility that some of our services combine request/response (i.e. choreography messages) and event-oriented (i.e notifications) message exchanges, a single infrastructure is a necessity. The OASIS Web Services Notification (WSN) family of specifications [23] illustrates how event-based programming can be introduced in SOA in a standardized way. The *WS-Base Notification specification*, which is the base document of WSN and often referred to as the *SOA notification pattern*, unifies the principles and concepts of SOA with those of event-based programming. Following this pattern, we can configure and deploy the different entities as event-oriented services. For instance, a *notification producer* may be deployed within the EFC. Depending on the granularity of the latter, this service may be responsible for detecting violations and creating notification messages instances while assuming the role of publisher of events. In the other side, *notification consumers* may be configured for internal monitoring applications.

An ESB[3] can be used here to mediate between request/response services and the new created event-oriented

[2]Policy Enforcement Point / Policy Decision Point
[3]Enterprise Service Bus

services helping for delivering notifications from producers to consumers. Nowadays, ESBs offer techniques for analyzing and controlling the complex series of interrelated events in order to allow for the implementation of both the SOA and the EDA concepts. The ESB may also provide additional added-value function such as transforming notification message content or logging notification messages in a relational database.

Furthermore, the *ESB gateway pattern* provides an XML firewall in addition to the desired gateway functions. Depending on security requirements, this ESB can be inside the Enterprise Demilitarized Zone (DMZ). Such a solution is well suited for serving on the boundaries of each organization as it is quite agile and flexible, thus permitting easy deployment of our designed components as services. However, this step needs more investigations and time and is part of our future work.

VI. Related work

The security of decentralized inter-organizational process has been an active research field in the last years. This section discusses different approaches and existing technical solutions in the field of access control, integrity of process execution, secure execution order and monitoring of choreographies.

Mendling et al. [13] presented an approach to integrate Role-Based Access Control (RBAC) and WS-BPEL[6] (Business Process Execution Language) at the meta-model level. Such approach, which is based on extending WS-BPEL with mappings to RBAC, can be used to automate steps of the role engineering process.

Thomas and Sandhu [19] proposed an extension of the RBAC called TBAC (Task-based Authorization Control). TBAC permits to grant or revoke permissions based on when tasks are scheduled in order to make related capabilities valid only for a specified duration. To make this possible, authorization policy had been extended with two new fields for enabling and disabling authorization at runtime, taking the form $< s, o, a, usage, authorization - step >$.

Using OrBAC (Organization based Access Control), Ayed et al. [2] suggested a dynamic decentralized approach to handle a security policy in inter-organizational workflows taking into account access and flow control. An OrBAC security rule takes the form $< type, organization, role, activity, view, context >$ where $type$ can be : $permission$, $prohibition$ or $obligation$. Like in TBAC, security rules here do not apply statically and their activations depend on contextual conditions. The algorithm defined in [2] shows how to use this model in a distributed WFMS environment. However, it needs further work on how to handle exchanging flows between different organizations.

In [9], the inter-organizational flow control is implemented with the same PEP/PDP mechanism. Indeed, the authorization specification is extended and takes the form of 7-tuple $< policy - id, s, o, a, l_{enable}, l_{disable}, state >$. The idea behind this is to enable and disable policies with respect to the current step in the choreography in order to ensure the correct execution order of the inter-organizational workflow. Each organization hosts a new component called Policy Generator or PGC which is responsible for generating a local policy during the workflow instantiation step. The generation depends on the workflow patterns. One drawback is that the $parallel$ pattern needs the generation of extra policies ($n * 2^{n-1}$ policies for n parallel branches). Vagts [20] extends this solution and proposes a decentralized framework for workflow execution that ensures exception safety and considers security issues.

Montagut and Molva [14] used *onion encryption* techniques to enforce the execution integrity of distributed workflows. The proposed solution ensures that the access to workflow data is performed with respect to the workflow execution plan and provides proofs of execution to the involved business partners. However, this technique may decrease performance and greatly increase the size of message.

Secure execution orders are also considered by Biskup et al. [4] supporting the case of a decentralized system by proposing a container structure with authentication mechanisms for data access. In other words, they introduce a framework for providing middleware support based on a process container, which calls dedicated middleware services to add support for security, persistence and reliable messaging. A similar approach is shown in [16]. In these two approaches the container is passed along the execution path of the workflow in order to verify the correct execution of inter-organizational choreographies. However, this technique does not address the exceptions that may occur, i.e. corruption or loss of the message or the container itself.

In [22], verification and validation techniques for Web service conversations and their choreographies are exposed. Nevertheless, the verification task is managed centrally and therefore not adapted to inter-organizational choreographies when there is no common trusted parties among all participants.

VII. Summary and future direction

Despite the numerous benefits attainable after adopting dynamic and flexible choreographies across collaborating parties, a major concern is the lack of security conventions after changing to an open and exposed environment. Without a coordinator, a decentralized control of these choreographies is a challenge. In this paper, we propose an architectural framework and a policy-based mechanism to verify the compliance of the overall sequence of inter-organizational choreography operations with the predefined process execution plan. Indeed, our approach aims to restrict the access to internal resources by filtering incoming calls taking into account the context of the call and the control-flow of the involved inter-organizational process. Further-

more, outgoing calls are also controlled aiming to prevent from divulging sensitive information in unexpected ways (i.e. without a coherent inter-organizational process context).

We first argue that our components should be policy-based as well as separated from the business application. This allows components to be managed centrally by a governance layer. The policies can be then audited, stored, traced, and historically managed.

The value added by this paper is that it deals with these issues in an event based way. In our approach, each message received or sent is associated to an event. Following the principle of Complex Event Processing (CEP), we use event patterns to filter and monitor desirable incoming and outgoing calls. This allows processing to be performed asynchronously which leads for a better responsiveness and less network traffic than the alternative request/response message exchanges. Furthermore, event processing, which becomes commonly used for monitoring and notification issues, helps us to get a dynamic and efficient decentralized way for errors notifications, along with providing execution traceability. At the time of writing this paper, there are no standards for event processing programming languages. Motivated by its notational simplicity, we have chosen RAPIDE-EPL to give an idea about how to implement our rules in practice. However, there are various languages and programming styles that can be used.

As future work, we intend to extend the architecture by additional event-driven components responsible for exceptions detection and reporting by sending external notification to the involved partners when an internal exception is raised. Moreover, and in order to further improve our message control, we are planning to extend the EFP table with additional fields imposing temporal and cardinality constraints.

REFERENCES

[1] E. Albek, E. Bax, G. Billock, K. M. Chandy, and I. Swett. An event processing language (epl) for building sense and respond applications. *IPDPS*, 2005.

[2] S. Ayed, N. Cuppens-Boulahia, and F. Cuppens. Managing access and flow control requirements in distributed workflows. *Computer Systems and Applications*, 2008.

[3] A. Baouab, O. Perrin, N. Biri, and C. Godart. Security meta-services orchestration architecture. *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific , vol., no., pp.28-33, 7-11*, Dec. 2009.

[4] J. Biskup, B. Carminati, E. Ferrari, F. Muller, and S. Wortmann. Towards secure execution orders for compositeweb services. *Web Services, 2007. ICWS*, 2007.

[5] G. Decker, O. Kopp, F. Leymann, and M. Weske. Bpel4chor: Extending bpel for modeling choreographies. *ICWS :296-303*, 2007.

[6] J. Evdemon et al. Web services business process execution language version 2.0. *Technical report, OASIS*, 2007.

[7] P. Grefen. Towards dynamic interorganizational business process management. *Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE*, 2006.

[8] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0. *W3C. Available from:*, 2005.

[9] F. Kerschbaum and P. Robinson. Security architecture for virtual organizations of business web services. *Journal of Systems Architecture, Volume 55, Issue 4, Secure Service-Oriented Architectures (Special Issue on Secure SOA)*, April 2009.

[10] N. Leavitt. Complex-event processing poised for growth. *Computer, vol. 42, no. 4, pp. 17-20*, Apr. 2009.

[11] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[12] Y. Magid, G. Sharon, S. Arcushin, I. Ben-Harrush, and E. Rabinovich. Industry experience with the ibm active middleware technology (amit) complex event processing engine. *In Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10). ACM, New York, NY, USA*, 2010.

[13] J. Mendling, M. Strembeck, G. Stermsek, and G. Neumann. An approach to extract rbac models from bpel4ws processes. *13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'04), pp. 81-86*, 2004.

[14] F. Montagut and R. Molva. Bridging security and fault management within distributed workflow management systems. *IEEE Transactions on Services Computing, pp. 33-48*, January-March, 2008.

[15] M. Papazoglou and P. Ribbers. *e-Business: organizational and technical foundations*. JohnWiley & Sons, Ltd., 2006.

[16] C. Rudolph, N. Kuntze, and Z. Velikova. Secure web service workflow execution. *Electronic Notes in Theoretical Computer Science, Volume 236, Proceedings of the 3rd International Workshop on Views On Designing Complex Architectures*, 2 April 2009.

[17] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *In: Proceedings of the IEEE 63, Nr. 9, 12781308*, 1975.

[18] S. Godik, T. Moses et al. Extensible access control markup language (xacml) version 1.0. *OASIS Standard*, February 18th, 2003.

[19] R. Thomas and R. Sandhu. Task-based authorization controls (tbac): A family of models for active and enterprise-oriented authorization management. 1997.

[20] H.-H. Vagts. Control flow enforcement in workflows in the presence of exceptions. *Masters thesis, TU Darmstadt*, December 2007.

[21] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's dance: A language for service behavior modeling. *OTM Conferences :145-162*, 2006.

[22] M. Rouached, W. Fdhila, and C. Godart. Web Services Compositions Modelling and Choreographies Analysis. *Int. J. Web Service Res.* , 2010.

[23] P. Niblett and S. Graham. Events and service-oriented architecture: The oasis web services notification specification. *IBM Systems Journal*, pages 869 –886, 2005.