# On the Impact of Concurrency for the Enforcement of Entailment Constraints in Process-driven SOAs

Thomas Quirchmayr and Mark Strembeck

Institute for Information Systems, New Media Lab, UZA II, WU Vienna, Austria
{firstname.lastname}@wu.ac.at

**Abstract.** A distributed business process is executed in a distributed computing environment. In this context, the service-oriented architecture (SOA) paradigm provides a mature and well understood framework for the integration of software services. Entailment constraints, such as mutual exclusion or binding constraints, are an important means to specify and enforce business processes in a SOA. However, the inherent concurrency of a distributed system may lead to omission and ordering failures. Such failures impact the enforcement of entailment constraints in a process-driven SOA. In particular, the impact of these failures as well as the corresponding countermeasures depend on the architecture of the respective process engine. In this paper, we discuss the impact of omission and ordering failures on the enforcement of entailment constraints in process-driven SOAs. In this context, we especially consider if the respective process engine acts as an orchestration engine or as a choreography engine.

## 1 Introduction

A business process describes a sequence of tasks which are executed sequentially or in parallel to achieve a business goal (see, e.g., [24]). In recent years, service-oriented architectures (SOA; see, e.g., [10, 13, 16, 17]) are increasingly used in the area of business process management. The SOA-paradigm is neutral from a technology point of view. However, Web services are a popular option to implement and deploy SOA services (see, e.g., [10, 17]). In this context, a *process-driven SOA* (see, e.g., [9]) is specifically built to support the definition, the execution, and monitoring of intra-organizational or cross-organizational business processes. In order to control and coordinate the services in a process-driven SOA, we have to ensure that the execution of the different services adheres to the process flow defined via the corresponding business process. In this context, a *process engine* is a software component that is able to control the process flow in a process-driven SOA.

In general, two architectural options for such process engines exist (see, e.g., [3, 4, 14, 18]). An *orchestration engine* acts as a central coordinator that communicates with different services and controls the process flow. If we use an orchestration engine, the services usually have no knowledge about their involvement in one or more business processes. In contrast, service choreography relies on collaborating *choreography engines*. Each of these choreography engines controls a certain part of the business process (e.g. a certain sub-process). Thus, in order to execute an entire business process the

This is an extended version of the paper published as: T. Quirchmayr, M. Strembeck: On the Impact of Concurrency for the Enforcement of Entailment Constraints in Process-driven SOAs , In: Proc. of the 10th International Workshop on Security in Information Systems (WOSIS), Angers, France, July 2013

In the extended version, we reinserted the text that we had to cut from the paper due to the page restrictions for the conference version.

different choreography engines (and thereby the services controlled via these choreography engines) need to be aware of their involvement into a larger process (to a certain degree). In this paper, we focus on the impact of omission and ordering failures in a process-driven SOA on the enforcement of entailment constraints. In particular, we discuss the differences that result from choosing an orchestration engine or choreography engines respectively.

### 1.1 Task-based Entailment Constraints

In a business process context, a *task-based entailment constraint* places some restriction on the subjects who can perform a task $x$ given that a certain subject has performed another task $y$. Entailment constraints are an important means to assist the specification and enforcement of business processes. Mutual exclusion and binding constraints are typical examples of entailment constraints (see, e.g., [6, 20, 25, 26]).

Mutual exclusion constraints can be subdivided into *static mutual exclusion* (SME) and *dynamic mutual exclusion* (DME) constraints. A SME constraint defines that two tasks (e.g. 'Order Supplies' and 'Approve Payment') must never be assigned to the same role and must never be performed by the same subject (to prevent fraud and abuse). This constraint is enforced at the type-level and thereby automatically applies to *all corresponding process instances*. A DME constraint is enforced at the instance-level by defining that two tasks must never be performed by the same subject in the *same process instance*. In contrast to mutual exclusion constraints, binding constraints define that two bound tasks must be performed by the *same* entity. In particular, a *subject-binding* (SB) constraint defines that the same individual who performed the first task must also perform the bound task(s). Similarly, a *role-binding* (RB) constraint defines that bound tasks must be performed by members of the same role but not necessarily by the same individual.

Most often, entailment constraint are defined in the context of a corresponding access control model. In recent years, role-based access control (RBAC) has developed into a de facto standard for access control in software-based systems. Process-related RBAC models define entailment constraints and corresponding access control polices in a business process context (see, e.g., [5, 21, 25]). In a process-driven SOA, the respective process engine must ensure the consistency of process-related RBAC models (see, e.g., [11]). In particular, it must enforce the entailment constraints and ensure the static (design-time) and dynamic (runtime) correctness of the corresponding models to prevent constraint conflicts or inconsistencies (see, e.g., [19, 20]).

Figures 1 and 2 show a simplified example of a business process that is executed via a process-driven SOA. Each task in such a process is performed by a certain subject. In order to perform the different tasks, the subjects invoke corresponding software services. If, for instance, there is a subject-binding between $t_a$ and $t_f$ and a DME constraint between $t_a$ and $t_e$ (see Figure 2), the respective process engine must ensure that for each process instance $p_i$, the corresponding task instances $t_{a_i}$ and $t_{f_i}$ are allocated to the same subject while the task instances $t_{a_i}$ and $t_{e_i}$ are allocated to different subjects (see, e.g., [19, 20]).

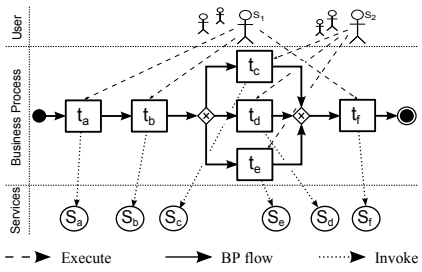



**Fig. 1.** Business Process Example in a SOA

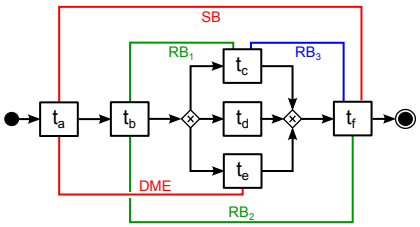



**Fig. 2.** Exemplary Entailment Constraints

### 1.2 Motivation

In the context of our work on the specification and enforcement of entailment constraints in business processes (see, e.g., [11, 19–21]), we implemented a corresponding runtime engine. Our runtime engine ensures the consistency of business processes with corresponding entailment constraints at the type-level (design-time) and at the instance-level (runtime). The source code of our implementation is available for download[1]. However, the deployment of such a process engine in a distributed system raises a number of challenges.

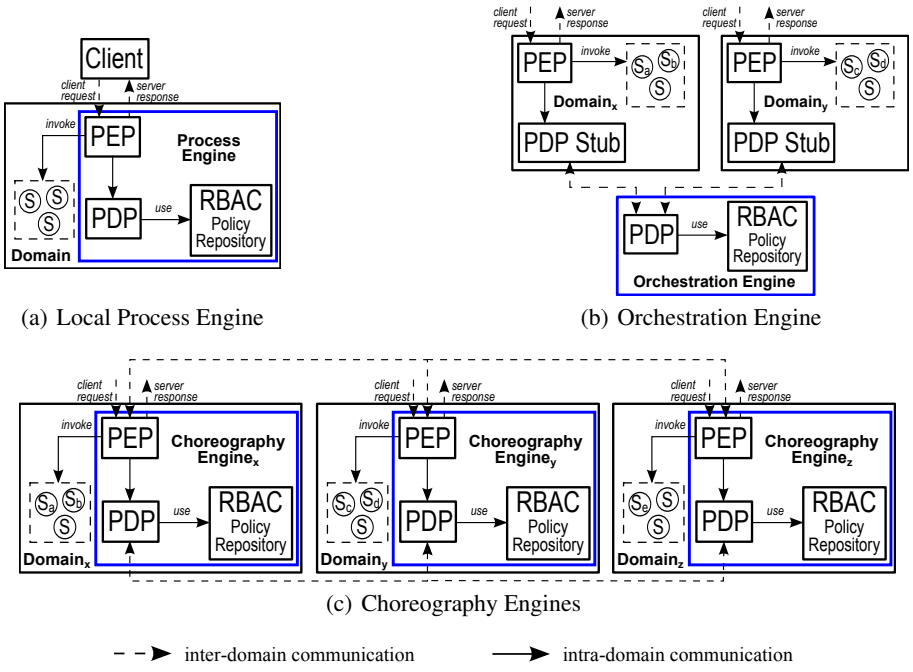
As mentioned above, the runtime enforcement of entailment constraints demands that the respective process engine correctly allocates the different task instances to corresponding subjects (see Section 1.1). In a process-driven SOA (see [9]), the allocation of tasks to subjects requires that the process engine and the services exchange corresponding messages. However, in a distributed system (see, e.g., [7]) ordering failures and omission failures may occur that impede the message exchange and thereby the enforcement of the entailment constraints. An omission failure occurs, if either a message is lost (e.g. due to a network failure) or if a machine crashes. Simplified, an ordering failure occurs if two messages are received in a different order by different receivers. In this context, orchestration engines and choreography engines apply different strategies to deal with such failures and to ensure the correct enforcement of entailment constraints.

The remainder of this paper is structured as follows: Section 2 gives an overview of different architectural options for the enforcement of entailment constraints in process-driven SOAs. Section 3 portrays different communication schemes to maintain task-allocation histories within a distributed business process. Sections 4 and 5 describe the impact of omission failures and ordering failures on the enforcement of entailment constraints respectively. Section 6 discusses the properties of different communication schemes with respect to the enforcement of entailment constraints. Section 7 discusses related work and Section 8 concludes the paper.

[1] http://wi.wu.ac.at/home/mark/BusinessActivities/library.html

## 2 Architectural Options for Enforcing Entailment Constraints

Figure 3 shows three basic options to enforce access control policies and entailment constraints in a process-driven SOA. Figure 3(a) shows the most simple option where the process engine and all services (in Figure 3 (Web) services are indicated by circles including a capital "S") that are controlled via this process engine are located at the same physical machine. This configuration has the advantage that the messages that need to be exchanged between the process engine and the services do not have to travel over a network. Furthermore, because all messages are exchanged on a single machine it is straightforward to maintain a local history log of all task allocations and access control decisions. However, in an actual SOA such a localized architecture is most often not a viable option (see, e.g., [2, 9, 10, 15]). Thus, Figures 3(b) and 3(c) sketch the architectures resulting from the use of an orchestration engine or interacting choreography engines respectively. Both options demand the exchange of messages over a network (in Figure 3 messages sent over a network are indicated by dashed lines).




(a) Local Process Engine

(b) Orchestration Engine

(c) Choreography Engines

**Fig. 3.** Three architectural options to enforce access control policies and constraints in a SOA

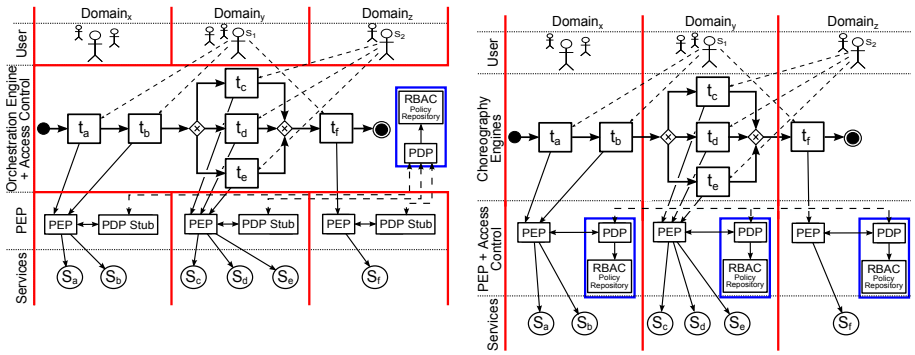
In case we use an *orchestration engine*, the services of each domain as well as the orchestration engine reside on a different physical machine (see Figure 3(b)). Thus, in order to make access control decisions and in order to allocate task instances in accordance with the corresponding entailment constraints, each domain must communicate

with the central orchestration engine. Because the orchestration engine acts as a central controller, it is able to keep a central process execution history. On the other hand, the orchestration engine is also a single point of failure which (in case of a system crash) will stop the entire system from working (see, e.g., [3, 23]).

If we use *choreography engines*, each engine can make local access control decisions and perform task allocations for the local services (see Figure 3(c)). However, in case a decision involves tasks that are constrained via mutual exclusion or binding constraints, the choreography engines must communicate to ensure the consistency of the entailment constraints (see Section 1.1). Moreover, because each choreography engine does only control a fragment of the entire business process, it is more difficult to maintain a complete process execution history (see, e.g., [7, 9]). As a result, the distributed nature of architectures relying on an orchestration engine or on choreography engines demands for a consideration of potential omission failures (see Section 4). As a result, the distributed nature of architectures relying on an orchestration engine or on choreography engines demands for a consideration of potential omission failures (see Section 4) and ordering failures (see Section 5).

## 3 Maintaining Task-allocation Histories in Process-driven SOAs

To allocate tasks in process-driven SOAs each process engine at least needs to know certain parts of the process history. For example, if the tasks $t_a$ and $t_e$ from Figure 4(b) are mutually exclusive, the choreography engine of domain $y$ needs to know which subject was allocated to an instance of $t_a$ to correctly allocate instances of $t_e$ (see, e.g., [19, 20]).




(a) Distributed Business Process Example with an Orchestration Engine

(b) Distributed Business Process Example with Choreography Engines

**Fig. 4.** Business Process Examples in a SOA based on different Process Engines

Figure 5 shows different communication schemes of entities participating in the distributed example business process from Figure 1. It illustrates the process flow and the

message exchange that is necessary to allocate task instances at runtime. An orchestration engine controls the entire business process and thus can allocate the task instances in accordance with the corresponding entailment constraints based on a central process history (see Figure 5(a)). For example, allocating $t_{a_i}$ requires an allocation-request sent from the orchestration engine ($OE$) to Domain x ($D_x$) which hosts the corresponding service $S_a$ (cf. Figure 4(a)). After a successful allocation of $t_{a_i}$, $D_x$ confirms the allocation. When the $OE$ receives the acknowledgement, the local history (indicated as $h_g$ in Figure 5(a)) is extended with the allocation information of $t_{a_i}$[2]. Maintaining history

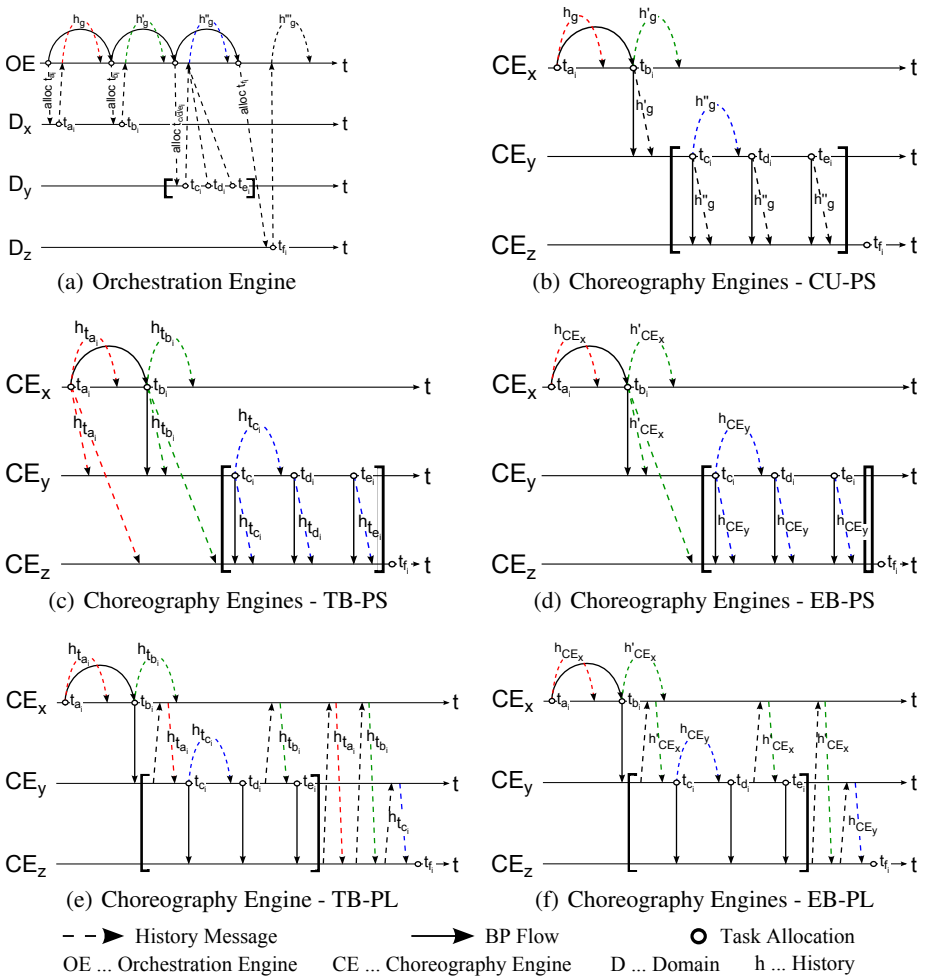



**Fig. 5.** History Management in Process-driven SOAs related to Figures 1, 2, and 4

[2] The square brackets in Figure 5(a) to 5(f) encompassing the task instances $t_{c_i}$, $t_{d_i}$ and $t_{e_i}$ indicate, that exactly one of them has to be allocated (see also Figures 1, 2 and 4).

information in a choreography engines architecture, on the other hand, is more complex as there does not exist a single history at a central location (see Figures 5(b) to 5(f)). Different approaches can be applied to obtain these information. First, it is possible to request the allocation information (which subject, executing a specific role, is allocated to a specific task instance) from the corresponding process engine before a task instance is to be allocated (*History Pull*). The second possibility is to send allocation information of task instances to certain process engines ex ante (*History Push*). Both approaches may operate at the task level (*task-based*), the process-engine level (*engine-based*), or on a global level (*cumulative*):

- A *Cumulative History Push (CU-PS)* communication scheme operates on a global level (i.e. it involves all choreography engines in a process-driven SOA) and extends the history with each access decision and task allocation. For example, in Figure 5(b) the allocation information of $t_{a_i}$ is recorded in the history log as $t_a$ is dynamically mutual exclusive to $t_e$ (see Figure 3(c)).
- In a *Task-based History Push (TB-PS)* communication scheme a choreography engine notifies the other engines as soon as a task is allocated to an executing-subject. For example, in Figure 5(c) choreography engine $CE_x$ notifies $CE_y$ immediately after $t_{a_i}$ was allocated. The message only contains allocation information of $t_{a_i}$ (indicated as $h_{t_{a_i}}$ in Figure 5(c)).
- In an *Engine-based History Push (EB-PS)* communication scheme the history push takes place when the process flow is passed from one choreography engine to another. For example, in Figure 5(d) $CE_x$ notifies $CE_y$ ($h'_{CE_x}$). This notification includes the history of all task allocations of $CE_x$.
- In a *Task-based History Pull (TB-PL)* communication scheme a choreography engine performs and on demand requests (pull) for the execution history of a particular task. For example, Figure 5(e) shows that the allocation history of each task is requested from the corresponding choreography engine (e.g., $CE_z$ requests the allocation information for $t_{a_i}$ and $t_{b_i}$, namely $h_{t_{a_i}}$ and $h_{t_{b_i}}$).
- In a *Engine-based History Pull (TB-PL)* communication scheme a choreography engine requests the entire execution history from another choreography engine (i.e. the history of all corresponding task instances). For example, in Figure 5(f) $CE_z$ requests the allocation history for $t_{a_i}$ and $t_{b_i}$, namely $h'_{CE_x}$, before allocating $t_{f_i}$.

Omission and ordering failures may impact the enforcement of entailment constraints in context of task allocation to a different extent, depending on the communication scheme used for administering historical task-allocation informations.

## 4 Omission Failures

An omission failure occurs, if either a message is lost (e.g. due to a network failure) or if a machine crashes (see, e.g., [7]).

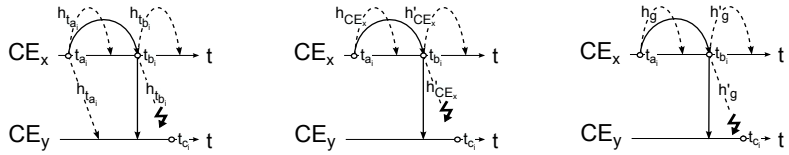
### 4.1 Lost Request or Lost Reply Messages

Depending on the process history scheme (see Section 3) a lost message may have different consequences on task allocation procedures (see Figures 6 to 8). Figure 6

shows lost messages in context of an orchestration engine architecture. The request from the orchestration engine to allocate a specific task to a subject or the response of the service domain $D_x$ may get lost. However, without confirming a task allocation the entire business process flow cannot be continued, because the allocation of subsequent tasks may depend on the respective process history (see Section 3).

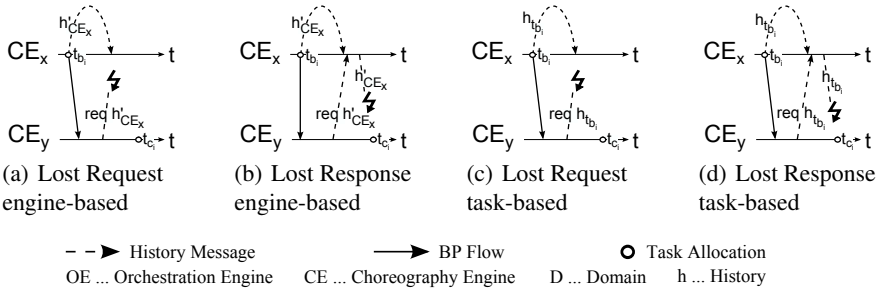

(a) Lost Request (b) Lost Response

**Fig. 6.** Orchestration Engine Message Loss




(a) Lost Notification task-based (b) Lost Notification engine-based (c) Lost Notification cumulative

**Fig. 7.** Choreography Engines Message Loss (History Push)




(a) Lost Request engine-based (b) Lost Response engine-based (c) Lost Request task-based (d) Lost Response task-based

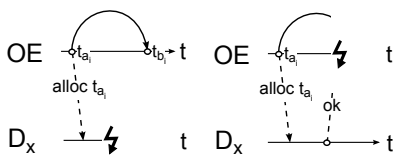
**Fig. 8.** Choreography Engines Message Loss (History Pull)

In a choreography engines architecture the main problem is the exchange of the process history between the choreography engines. If a choreography engine cannot access the process history, it cannot allocate tasks that must adhere to entailment constrains. Figures 7(a) and 7(b) sketch the loss of a task-based respectively engine-based history push (indicated as $h_{t_{b_i}}$ and $h'_{CE_x}$ respectively). Figure 7(c) shows the loss of a cumulative history push stopping the entire business process (see Sections 1.1 and 3).

Figure 8 shows lost allocation history request and response messages based on task-based and engine-based history pull. If a history request (see Fig. 8(a)) or the respective

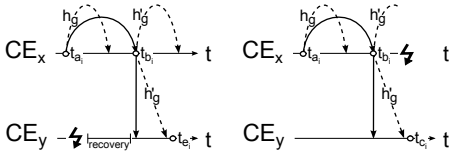
response (see Figure 8(b)) is lost, the entire business process may stop – which means that all tasks that have a binding or a mutual exclusion constraint to preceding tasks, cannot be allocated.

### 4.2 Sender and Receiver Crash



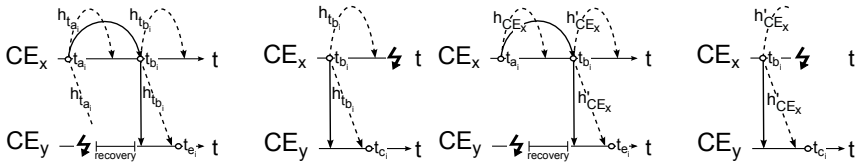

(a) Receiver Crash (b) Receiver Crash

**Fig. 9.** Orchestration Engine



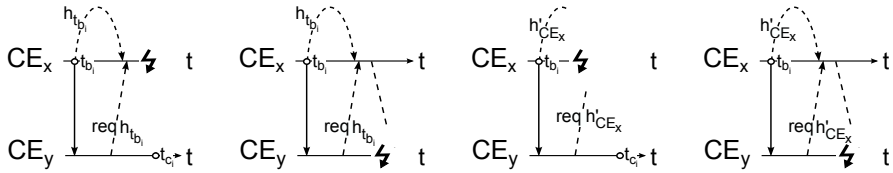

(a) Receiver Crash (b) Sender Crash

**Fig. 10.** Choreography Engines - Cumulative




(a) Receiver Crash task-based (b) Sender Crash task-based (c) Receiver Crash engine-based (d) Sender Crash engine-based

**Fig. 11.** Choreography Engines - History Push




(a) Receiver Crash task-based (b) Receiver Crash task-based (c) Sender Crash engine-based (d) Receiver Crash engine-based

History Message BP Flow Task Allocation
OE ... Orchestration Engine CE ... Choreography Engine D ... Domain h ... History

**Fig. 12.** Choreography Engines - History Pull

Figures 9(a) and 9(b) show the crash of a receiver in an orchestration engine architecture. In particular, the crash occurs while trying to allocate a subject to task instance $t_{a_i}$. In both cases the allocation fails and the business process cannot be continued (cf. Sections 1.1 and 3). Figures 10 and 11 depict crashes in a choreography engines architecture based on history push. Figures 10(b), 11(b) and 11(d) show that a sender crash may not impact the allocation of subsequent tasks (as long as another choreography

engine controls the current business process flow) if the corresponding process history was previously delivered from $CE_x$ to $CE_y$. A receiver crash may lead to difficulties (see Figures 10(a), 11(a) and 11(c). In particular, if the receiver (in the example: $CE_y$) is not able to recover (see, e.g., [7]) before the process flow is passed from $CE_x$ to $CE_y$, the process execution is stopped. On the other hand, if the receiver recovers in time, it is possible to allocate $t_{e_i}$ if we use a cumulative or engine-based history push scheme (see Section 3). In case of a task-based history push Figure 11(a) shows that $h_{t_{a_i}}$ could not be delivered and thus $t_{e_i}$ may not be allocated.

A crash in a choreography engines architecture that uses history pull scheme may also lead to task allocation problems. For example, in Figure 12(a) a crash of $CE_x$ may interrupt the process flow because $CE_y$ cannot allocate $t_{c_i}$ without first receiving the process history from $CE_x$. In a similar way, the process flow is interrupted if $CE_y$ crashes after $CE_x$ has sent the process history (see Figures 12(b) and 12(d)). However, in the example from Figure 12(c) $CE_x$ may crash after it sent the process history to $CE_y$ – in such as scenario, the process flow will not be interrupted. As it is basically neither possible for the receiver nor for the sender to clearly distinguish a crash from lost messages it may be advantageous to establish a so called "heartbeat" scheme (see, e.g., [8]).

## 5 Ordering Failures

Because distributed systems usually cannot rely on a global physical clock nor on exactly synchronized local clocks, it is not trivial to order events (e.g., sending of a message, reception of a message, task execution) that occur on different machines in a distributed system (see, e.g., [7, 12]). In the following, ordering failures are shown in choreography engines using task-based history push exemplary, as this combination may be most impacted by these failures because it requires most messages to be sent. The discussion analogously applies to the other history schemes from Figure 5.

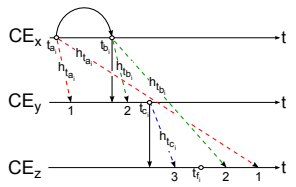
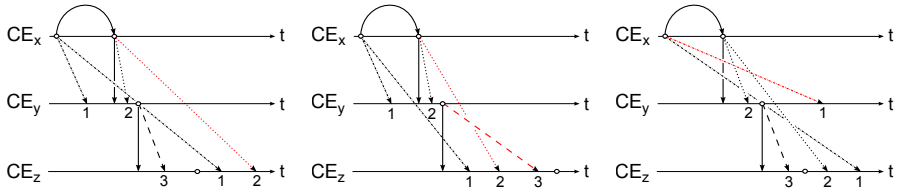



**Fig. 13.** Ordering Failures within a Choreography Engines Architecture related to Figure 5(c)

Figure 13 depicts ordering failures that may occur in a choreography engines architecture using task-based history push without any message ordering mechanism. It shows that history notifications are delivered in a different chronological sequence they were sent (e.g., $h_{t_{a_i}}$ is delivered to $CE_z$ after $h_{t_{b_i}}$ and $h_{t_{c_i}}$). Without any ordering mechanism it is not possible to determine to which subject $t_{f_i}$ is to be allocated.



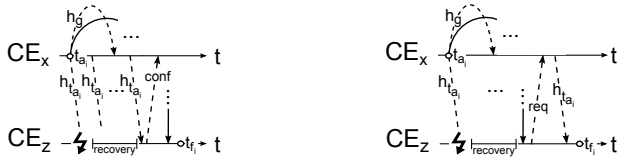

(a) FIFO Ordering (b) Causal Ordering (c) Total Ordering

**Fig. 14.** FIFO, Causal and Total Message Ordering related to Figure 13

However, different well-known ordering schemes exist. *FIFO Ordering* relates to the sequence of messages sent from a specific sender perspective (see, e.g., [7, 12]). If an event $e_1$ happended before another event $e_2$ (written as $e_1 \rightarrow e_2$) then $e_1$ may causally affect $e_2$. A *causal ordering* scheme essentially enforces a "global FIFO" ordering on all messages (rather than on messages from one particular sender only). *Total ordering* defines that messages are delivered in the same order from all participating and correct receivers, but not at all that the messages are in the same sequence they were issued by the sender. *FIFO-total ordering* guarantees message delivery under consideration of FIFO as well as total ordering whereas *Causal-total ordering* arranges messages in causal and total order (see, e.g., [7]).

## 6 Discussion

A history push approach relies on a synchronous request-reply protocol and thus always expects (and waits for) a reply, in our case the history information of another choreography engine. History push approaches also need a mechanism to ensure the delivery of allocation histories. Therefore, basically two different solutions exist. The first one is to introduce a 'confirmation'-message from the receiving choreography engine to the sending choreography engine after receiving the history information (see Figure 15(a); indicated via the word 'Conf' in Table 1). The history message is sent until it is confirmed (asynchronous).




(a) Task-based History Push with Receiver Confirmation

(b) Task-based History Push with Pull

**Fig. 15.** Approaches to ensure History Message Delivery in a Choreography Engines Architecture

The second option is to combine push and pull (see Figure 15(b)). All history-related messages are pushed to the choreography engines exactly once (either task-based, engine-based, or cumulative). If a history record is not available at the time a specific task is to be allocated, the corresponding information is explicitly requested from the respective choreography engine (pull). On the one hand, a push approach with a confirmation message can ensure an accurate delivery of history information (even in the case of omission failures). On the other hand, a push-pull approach possibly minimizes the amount of history-related messages that need to be sent.

Table 1 shows the properties of the different communication schemes related to five performance categories. The performance categories have the following meaning:

- *Timeliness* relates to an accurate provision of history-related information (is a history information available when needed or does it have to be explicitly requested).
- *Number of Messages* indicates the amount of history-related messages to be exchanged between choreography engines (e.g. request, reply, notification).
- *Size of Messages* relates to the size of the message payload (single task-allocation record or a complete engine-specific history respectively a complete process history).
- *Impact of Failure* is the degree to which the execution of the entire business process is affected (interference of a part of the business process or the entire business process).
- *Effort of Ordering* refers to the need to implement an ordering mechanism (see Section 5).

**Table 1.** Performance Evaluation of Task-allocation History Maintenance Approaches
(++ Very High Performance, - - Very Low Performance, ○ Average Performance)

| **Performance Category** | **Choreography Engines** | | | | | **Orchestration Engine** |
|---|---|---|---|---|---|---|
| | TB-PS | EB-PS | CU-PS | TB-PL | EB-PL | |
| Timeliness (Conf) | + (++) | + (++) | + (++) | - - | ○ | ○ |
| Number of Messages (Conf) | - (- -) | ○(○) | ++ (+) | - | ○ | - - |
| Size of Messages | ++ | ○ | - - | ++ | ○ | ++ |
| Impact of Omission Failure (Conf) | ++ (++) | + (+) | ○(○) | - | ○ | - - |
| Effort of Ordering | - - | - - | ++ | ++ | ++ | ++ |

Related to the communication schemes mentioned in Table 1 their properties can be interpreted as follows:

- *Task-Based History Push.* All history push communication schemes provide historical information in a timely manner. A push-pull approach may demand an additional request message before allocating a task. However, a confirmation message ensures accurate delivery of history information. This communication scheme requires one message (history) respectively two messages (history and confirmation)

for each constraining task at least. Moreover, the payload of each message includes a single-task allocation record only. Because process control is distributed and the allocation histories are sent in advance for each constraining task, the impact of omission failures is small. Moreover, we require an ordering mechanism to ensure the correct submission of multicast messages. A choreography engine multicasts its entire allocation history (multiple task allocation records) to all choreography engines that control constrained tasks. In this scheme, fewer messages are sent (one history and possibly a confirmation message) but the payload increases (task allocation history of an entire choreography engine). Also the impact of omission failures increases as the delivery of a history may be more time-critical. Moreover, we require an ordering mechanism to ensure the correct submission of multicast messages.

- *Cumulative History Push.* This scheme requires the smallest number of messages to be sent. The history message contains all previous task allocation records of the respective process. Because as single history is passed between the choreography engines, its delivery is still more time-critical. However, as multicasting is not necessary, we do not need to implement an ordering mechanism.
- *Task-Based History Pull.* In this scheme, the respective choreography engine has to request the allocation history of the constraining task(s) before allocating a constrained task. Similar to task-based history push the message size is small (a request respectively a response consisting of a single task-allocation record). As the allocation of a constrained task heavily depends on the communication between choreography engines, an omission failure may have significant effects. However, as multicasting is not necessary there is no need to implement an ordering mechanism.
- *Engine-Based History Pull.* In this scheme, each choreography engine requests engine-based allocation histories when allocating its first constrained task. Similar to engine-based history push, the number of messages decreases but their size increases compared to task-based history exchange. Omission failures may delay task allocation for the allocation of the first constrained task. However, as multicasting is not necessary there is no need to implement an ordering mechanism.
- *Orchestration Engine.* This scheme maintains the entire business process history locally but has to communicate with the different remote services. As there is no need to exchange a history, the messages are allocation requests of small size (a single request and a respective confirmation for each task to be allocated). An orchestration engine architecture is most impacted by omission failures. In case the orchestration engine suffers a crash, the execution of the entire business process freezes. Each crashed domain, hosting a task to be allocated next, also stops at least a part of the business process from working. However, as multicasting is not necessary there is no need to implement an ordering mechanism.

In addition to the assessment shown in Table 1, the following three interrelated determinants have to be considered in order to choose a proper process engine architecture: the *number of constrained tasks per business process* (degree of constraint; DOC), the *number of participants* in the business process (degree of distribution; DOD) and the *number of business process control transitions* between different participants in a busi-

ness process instance (degree of networking; DON). According to these characteristics and the corresponding performance categories, we can choose the approach that best fits a particular SOA. For example, a business process with a high DOC, a high DOD, and a high DON may best be handled with a choreography engines architecture using an engine-based history push approach with confirmation. On the other hand, if our focus is on minimized size of messages and minimal costs for implementing an ordering mechanism, an orchestration engine architecture may be a better choice.

## 7 Related Work

Several approaches address the enforcement of entailment constraints during task allocation. In [22], Tan et al. present an approach for constraint specification within a workflow authorization schema. Furthermore they define a set of consistency rules for constraints to prevent inconsistencies and ambiguities between constraints. Bertino et al. [5], propose a language for expressing entailment constraints and algorithms to check the consistency of these constraints while assigning roles and users to workflow tasks. Similarly, Schefer et al. [19] discuss resolution strategies for conflicts of process-related mutual-exclusion and binding constraints before these conflicts cause an inconsistent RBAC configuration. Xu et al. [27] consider concurrency in access control decisions through the development of XACML-ARBAC, a language to resolve the concurrency problem. However, they focus on the administration of session-aware RBAC models and do not discuss the problems of enforcing entailment constraints in a distributed environment. In particular, they assume fail-save participants and processes, reliable communication, as well as a centralized workflow coordinator. Ayed et al. [1] discuss the deployment of workflow security policies for inter-organizational workflow. However, the approach also assumes fail-save hard- and software. Our work is complementary as it discusses the enforcement of entailment constraints in distributed systems at runtime considering omission and ordering failures. In particular, we consider omission and ordering failures that may occur in a process-driven SOA.

## 8 Conclusion

In this paper, we discussed the impact of omission and ordering failures on the enforcement of entailment constraints in process-driven SOAs. Because the enforcement of entailment constraints relies on the availability of a process history, we observe different history schemes and examine the impact of failures on architectures that use an orchestration engine or choreography engines respectively. This paper was inspired by our work on the specification and enforcement of entailment constraints in business processes (see, e.g., [11, 19–21]) and the implementation of a corresponding runtime engine[3].

In recent years, we see an increasing interest in process-aware information systems in both research and practice. In this context, an increasing number of existing and future systems will have to be extended with respective consistency checks. The discus-

[3] available from: http://wi.wu.ac.at/home/mark/BusinessActivities/library.html

sion from this paper can help to address the challenges that result from the deployment of a process engine in a distributed system.

## References


1. S. Ayed, N. Cuppens-Boulahia, and F. Cuppens. Deploying security policy in intra and inter workflow management systems. *2012 Seventh International Conference on Availability, Reliability and Security*, 0:58–65, 2009.
2. J. Bacon and K. Moody. Toward open, secure, widely distributed services. *Commununication of ACM*, 45(6):59–64, June 2002.
3. A. Barker, C. D. Walton, and D. Robertson. Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152–166, 2009.
4. A. Barros, M. Dumas, and P. Oaks. Standards for web service choreography and orchestration: Status and perspectives. In *Proceedings of the Workshop on Web Services Choreography and Orchestration for Business Process Management*, 2005.
5. E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, Feb. 1999.
6. F. Casati, S. Castano, and M. Fugini. Managing workflow authorization constraints through active database technology. *Information Systems Frontiers*, 3(3):319–338, Sep 2001.
7. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design (5th Edition)*. Addison Wesley, May 2011.
8. I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 170–179, New York, NY, USA, 2001. ACM.
9. C. Hentrich and U. Zdun. *Process-Driven SOA: Patterns for Aligning Business and IT*. CRC Press, Taylor and Francis, 2012.
10. M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, Jan. 2005.
11. W. Hummer, P. Gaubatz, M. Strembeck, U. Zdun, and S. Dustdar. An integrated approach for identity and access management in a SOA context. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 21–30, New York, NY, USA, 2011. ACM.
12. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of ACM*, 21(7):558–565, July 1978.
13. G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM Transactions on Web*, 4(1):2:1–2:33, Jan. 2010.
14. N. Milanovic and M. Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, Nov. 2004.
15. G. Neumann and M. Strembeck. Design and implementation of a flexible RBAC-service in an object-oriented scripting language. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, CCS '01, pages 58–67, New York, NY, USA, 2001. ACM.
16. M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, July 2007.
17. M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11):38–45, Nov. 2007.
18. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.

19. S. Schefer, M. Strembeck, J. Mendling, and A. Baumgrass. Detecting and resolving conflicts of mutual-exclusion and binding constraints in a business process context. In *Proceedings of the 19th International Conference on Cooperative Information Systems (CoopIS)*, volume 7044 of *Lecture Notes in Computer Science (LNCS)*, pages 329–346, Berlin, Heidelberg, 2011. Springer-Verlag.
20. M. Strembeck and J. Mendling. Generic algorithms for consistency checking of mutual-exclusion and binding constraints in a business process context. In *Proceedings of the 18th International Conference on Cooperative Information Systems (CoopIS)*, volume 6426 of *Lecture Notes in Computer Science (LNCS)*, pages 204–221, Berlin, Heidelberg, 2010. Springer-Verlag.
21. M. Strembeck and J. Mendling. Modeling process-related RBAC models with extended UML activity models. *Information & Software Technology*, 53(5):456–483, 2011.
22. K. Tan, J. Crampton, and C. A. Gunter. The consistency of task-based authorization constraints in workflow systems. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, pages 155–170, Washington, DC, USA, 2004. IEEE Computer Society.
23. M. V. Tripunitara and B. Carbunar. Efficient access enforcement in distributed role-based access control (RBAC) deployments. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 155–164, New York, NY, USA, 2009. ACM.
24. W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business process management: A survey. In W. M. P. van der Aalst and M. Weske, editors, *Business Process Management*, volume 2678 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2003.
25. J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC - a workflow security model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, 12:2003, 2003.
26. C. Wolter and A. Schaad. Modeling of task-based authorization constraints in BPMN. In *Proceedings of the 5th international Conference on Business Process Management*, BPM'07, pages 64–79, Berlin, Heidelberg, 2007. Springer-Verlag.
27. M. Xu, D. Wijesekera, X. Zhang, and D. Cooray. Towards session-aware RBAC administration and enforcement with XACML. In *Proceedings of the 10th IEEE International Conference on Policies for Distributed Systems and Networks*, POLICY'09, pages 9–16, Piscataway, NJ, USA, 2009. IEEE Press.