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, WU Vienna, Vienna, Austria

Abstract. Entailment constraints, such as mutual exclusion or binding constraints, are an important means to specify and enforce business processes. However, the inherent concurrency of a distributed system may lead to omission. 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 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 *process-driven SOA* (see, e.g., [6]) 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., [2, 10, 11]). 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 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).

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 (see Figure 1 and 2). 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., [4, 12, 15]).

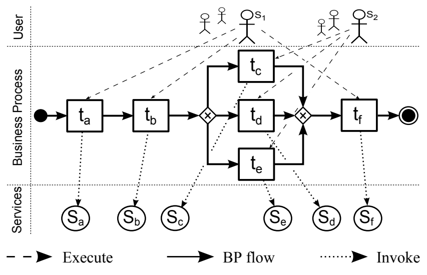


Fig. 1. Business Process Example in a SOA.

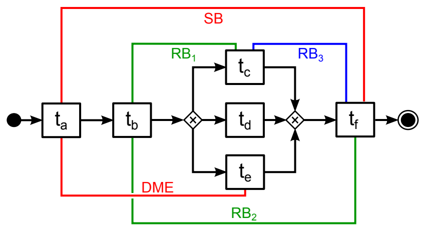


Fig. 2. Exemplary Entailment Constraints.

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). 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.

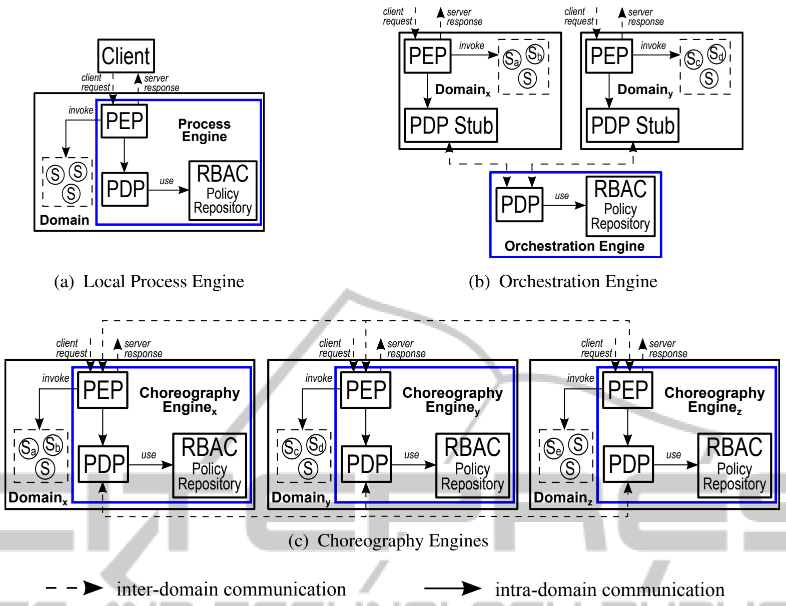
Most often, entailment constraint are defined in the context of a corresponding access control model. Process-related RBAC models define entailment constraints and corresponding access control polices in a business process context (see, e.g., [3, 13, 15]). In a process-driven SOA, the respective process engine must ensure the consistency of process-related RBAC models (see, e.g., [8]).

1.2 Motivation

In a process-driven SOA (see [6]), 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., [5]) 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. 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.

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

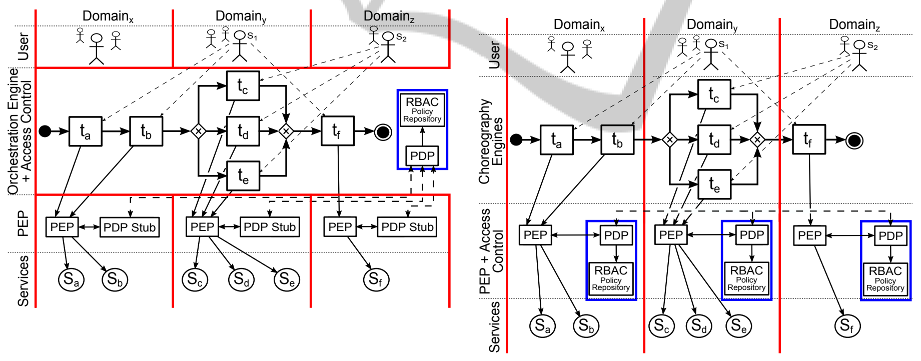


(a) Local Process Engine

(b) Orchestration Engine

(c) Choreography Engines

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



(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.

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., [6, 7]). 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).

3 Maintaining Task-allocation Histories in Process-driven SOAs

Figure 4(a) shows that the allocation of 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. In a choreography engines architecture the tasks are allocated locally (see Figure 4(b)). Additionally, to allocate tasks in process-driven SOAs to corresponding entailment constraints, each process engine at least needs to know certain parts of the process history. 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)). [1]. Maintaining history 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. 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. 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. 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.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).

4 Omission Failures

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

[1] 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).

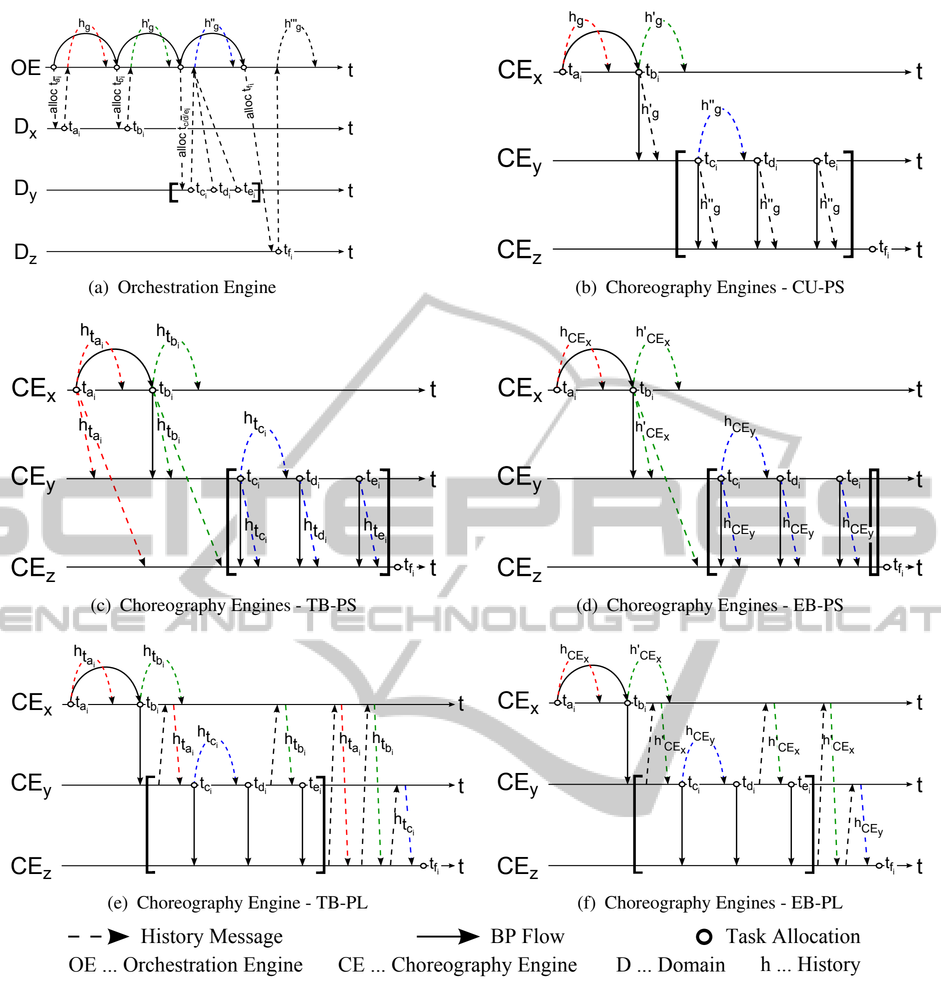
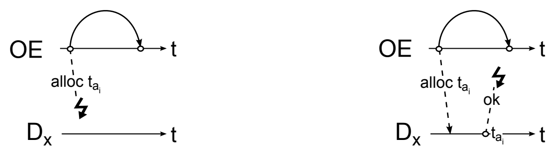

(a) Orchestration Engine
(b) Choreography Engines - CU-PS
(c) Choreography Engines - TB-PS
(d) Choreography Engines - EB-PS
(e) Choreography Engine - TB-PL
(f) Choreography Engines - EB-PL
History Message
BP Flow
Task Allocation
OE ... Orchestration Engine
CE ... Choreography Engine
D ... Domain
h ... History

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

entire business process flow cannot be continued, because the allocation of subsequent tasks may depend on the respective process history (see Section 3).

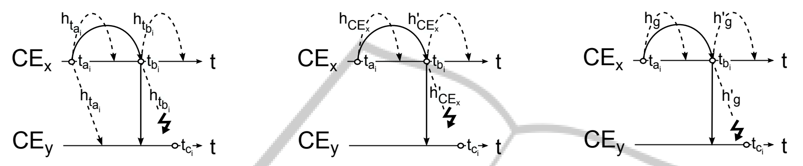
In a choreography engines architecture the main problem is the exchange of the process history between the choreography engines. If a choreography engine cannot access the process history, it cannot allocate tasks that must adhere to entailment constraints. Figures 7(a) and 7(b) sketch the loss of a 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.



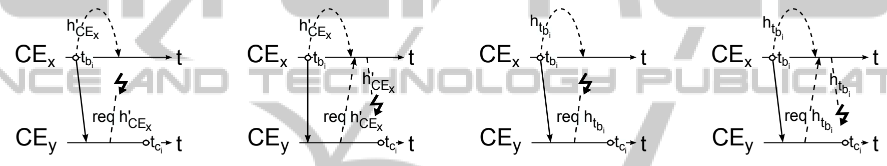
(a) Lost Request (b) Lost Response

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

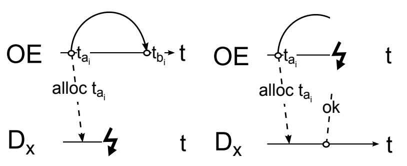
- - ► History Message ——► BP Flow ○ Task Allocation
OE ... Orchestration Engine CE ... Choreography Engine D ... Domain h ... History

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

4.2 Sender and Receiver Crash

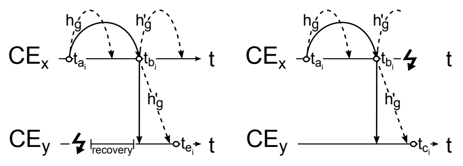
Figures 9(a) and 9(b) show the crash of a receiver in an orchestration engine architecture. In particular, the crash occurs while trying to allocate a subject to task instance t_{a_i}. In both cases the allocation fails and the business process cannot be continued (cf. Sections 1.1 and 3). Figures 10 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., [5]) 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



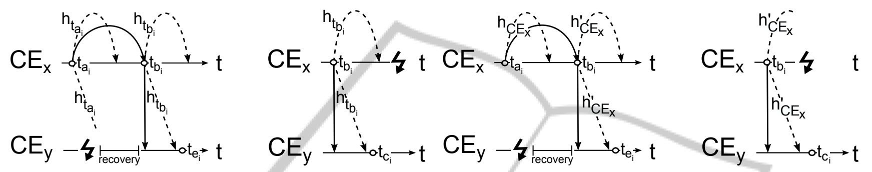
(a) Receiver Crash (b) Receiver Crash

Fig. 9. Orchestration Engine.



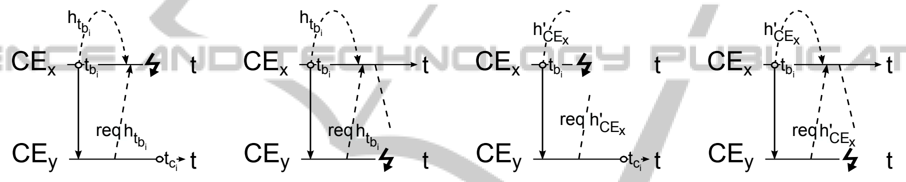
(a) Receiver Crash (b) Sender Crash

Fig. 10. Choreography Engines - Cumulative.



(a) Receiver Crashtask-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 o Task Allocation
OE ... Orchestration Engine CE ... Choreography Engine D ... Domain h ... History

Fig. 12. Choreography Engines - History Pull.

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., [5]).

5 Discussion

A *Task-Based History Push* 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 (see, e.g., [9]). 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. A *Cumulative History Push* 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. In the *Task-Based History Pull* 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. In a *Engine-Based History Pull* 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. In an *Orchestration Engine* the entire business process history is maintained locally but it 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, 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 business 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.

6 Related Work

Several approaches address the enforcement of entailment constraints during task allocation. In [14], 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. Xu et al. [16] 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 failures. In particular, we consider omission failures that may occur in a process-driven SOA.

7 Conclusions

In this paper, we discussed the impact of omission 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., [8, 12, 13]) and the implementation of a corresponding runtime engine[2]. 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 discussion 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. A. Barker, C. D. Walton, and D. Robertson. Choreographing web services. IEEE Transactions on Services Computing, 2(2):152–166, 2009.
3. 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.
4. F. Casati, S. Castano, and M. Fugini. Managing workflow authorization constraints through active database technology. Information Systems Frontiers, 3(3):319–338, Sep 2001.

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

5. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. Distributed Systems: Concepts and Design (5th Edition). Addison Wesley, May 2011.
6. C. Hentrich and U. Zdun. Process-Driven SOA: Patterns for Aligning Business and IT. CRC Press, Taylor and Francis, 2012.
7. M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. IEEE Internet Computing, 9(1):75–81, Jan. 2005.
8. 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, USA, 2011.
9. L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communication of ACM, 21(7):558–565, July 1978.
10. N. Milanovic and M. Malek. Current solutions for web service composition. IEEE Internet Computing, 8(6):51–59, Nov. 2004.
11. C. Peltz. Web services orchestration and choreography. IEEE Computer, 36(10), 2003.
12. 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, 2010.
13. M. Strembeck and J. Mendling. Modeling process-related RBAC models with extended UML activity models. Information & Software Technology, 53(5):456–483, 2011.
14. 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, USA, 2004.
15. 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.
16. 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, USA, 2009.