Distributed Execution of S-BPM Business Processes

Erwin Aitenbichler, Stephan Borgert, and Max Mühlhäuser

Technische Universität Darmstadt, Hochschulstrasse 10, 64289 Darmstadt, Germany

Abstract. Subject-oriented business process management (S-BPM) introduces a new technique for process modeling that emphasizes the importance of the actors in business processes (subjects) and gives a balanced consideration to subjects, their actions, and goals. Because of the formal foundation and the clear declaration of subjects, S-BPM allows the distributed modeling and execution of processes, without losing the capability to verify the compatibility of processes. Executing cooperating processes in a distributed system also poses new requirements to the communication middleware, which is responsible for routing messages from one process instance to a remote peer process instance. In this paper, we describe an engine to execute S-BPM process choreographies. It is based on subject-oriented process modeling and a publish/subscribe middleware as communication basis. Our process engine also runs on mobile devices.

1 Introduction

Internet-based marketplaces for mobile phone end-user applications ("apps") have been very successful recently. As a second trend, we observe the increased adoption of service-oriented architectures in enterprises. Such architectures make systems modular and components become interchangeable. The combination of both – marketplaces and services – promises to enable an *Internet of Services*, which offers marketplaces where services become tradable goods. The basis for such an Internet of Services is currently developed in the large-scale Theseus Programme [1].

The Internet of Services gives businesses the opportunity to outsource certain business functions to external contractors by using external services, instead of implementing them in-house. From the business process perspective, the consequence of outsourcing is that a subprocess of the overall internal business process is moved to an external contractor. This subprocess is then implemented by the external service used. However, in an open service market, where anybody is allowed to offer services, it seems natural that there will be multiple offers for services providing the same functionality.

Hence, the services offered on the market will be different in many details, such as their quality and how their internal processes are realized. In many cases, the process implemented by an offered service will not exactly match the subprocess a customer asks for. As long as the remaining internal process and the outsourced subprocess fulfill *interaction soundness* [2], they can be composed. Consequently, it is vitally important that the parties interacting in a B2B scenario ensure that their processes are compatible with each other. It is desirable that this compatibility check is performed on the models during design or composition time, thereby detecting any incompatibilities before some process instance actually fails at runtime.

As a direct consequence of the Internet of Services, the distributed execution of business processes also increases in importance. By distributed execution of a business process we mean that every process participant may use her own process execution engine. The overall process is then executed by interconnecting multiple engines. These engines may be used by different enterprises, but there may also be multiple engines within the same enterprise. The engine may also run on the mobile device of an employee, giving her the opportunity to work on processes during times without a network connection. This paper addresses the following two aspects of distributed process execution:

- the composition-time compatibility check of services and
- the execution-time message routing between process engines

1.1 Requirements

For the distributed execution of a process, two important prerequisites are needed: A *suitable process modeling* technique and a *flexible communication platform*. We expect that the process modeling technique can fulfill the following requirements:

- It should be possible to split a process up into subprocesses describing the work of each process participant separately.
- If a process is modeled in a distributed manner, it should be possible to perform the same model checks regarding the communication between interacting partners, as if it would be modeled as a whole.

In order to execute the process in a distributed manner, it must first be clear which activities are performed by which actor. Subject-oriented process modeling introduces an approach that gives balanced consideration to the actors in business processes (persons and systems as subjects), their actions (predicates), and their goals or the subject matter of their actions (objects) [3, 4]. Every one of them receives and delivers information by exchanging messages. Humans, e.g., exchanges emails, office documents, or voice messages.

Because subjects are modeled separately, it is easy to split up a process into its subprocesses for all its subjects. It is also possible that subjects are modeled by different enterprises at different places. Because the formal framework offers the functionality to hide all internal communication and communication with other third parties for a particular interaction, companies only have to expose minimum knowledge about their internal processes.

1.2 Contributions

In this work we make the following contributions:

- We show that subject-orientation and the process modeling language Parallel Activities Specification Scheme (PASS) [5] provide a good basis for the distributed modeling and execution of business processes. Parties do not have to disclose their internal processes and can still ensure the compatibility of their processes.

– We describe the *process embedding* concept and its implementation in a distributed system. It allows to reduce the number of process variants that would otherwise have to be modeled and maintained explicitly.
– We show how the publish/subscribe abstraction can be used for the flexible message routing needed in distributed process execution.

The rest of the paper is organized as follows. In section 2 we discuss related work. Subject-oriented BPM and the modeling language PASS are introduced in section 3. Next, section 4 describes how PASS can be mapped to its formal foundation CCS. The execution of PASS processes and the separate modeling of the embedding information is presented in section 5. In section 6 we show how publish/subscribe can be applied to message routing in distributed process execution. The implementation is described in section 7. Finally, the paper is concluded in section 8.

2 Related Work

2.1 Process Description Languages

The mainstream process description languages used today lack a formal foundation and well-defined semantics (e.g., EPC, BPMN). This impedes to perform verifications on the models or to directly execute the models. Execution-oriented languages (e.g., BPEL, XPDL) are also not suitable for verification, because they hardly allow the description of choreographies. Consequently, in practice business experts often start with a business-oriented model (BPMN), which is then one-shot transformed into an execution-oriented model (BPEL) by engineers. In contrast to using multiple languages, PASS can be used throughout the whole process lifecycle. It has a formal basis suitable to perform verifications and is executable at the same time.

Several approaches are working on the execution level and extend the functionality of the BPEL standard by using proxy services or additional annotations or descriptions. A representative work is described in [6], where the authors introduce the VxBPEL language, which is an extension of BPEL by incorporating variability. It enables rebinding of services during runtime, substitution of service for optimizing purposes or in case of in sudden unavailable services. In contrast to our approach, choreographies are not supported. In addition, services provided by humans are not considered and there is a shortage of formal verification techniques.

2.2 Formal Methods for Verifying Interaction Soundness

Process algebras like the π-calculus provide strong means for modeling concurrent systems like service compositions and are based on formal terms. Choreography modeling, refining, and clustering are inherently supported. In addition, a rich theory to analyze processes for equivalence is provided and also the capability to perform reasoning on system properties and to verify process behavior. For this reason, current research efforts in this area focus mainly on approaches for formal verification of services and business process. Work on compliance and compatibility checks investigate the issue

of when a service can be replaced by another one [7, 8]. This is necessary when a service of a process fails during runtime or for finding redundant services. COWS [9] and SOCK [10] are designed for the purpose of automatic service composition. Furthermore, process algebras are often combined with other formalisms in order to be able to specify more aspects of a service in a formal manner. E.g., some extensions exist, that combine the π-calculus with ontologies and formal logics [11, 12] to describe non-functional properties and access control policies. While these formal approaches are also capable of formal verification and matchmaking, they usually do not consider other aspects, such as the execution of the models, or the seamless integration of human services. In this work we use CCS, which can be considered as a subset of the π-calculus. The additional concepts of the π-calculus, like mobility of channels, were not required here.

A Petri Net is a formal language for modeling concurrent systems and has been widely accepted as formal foundation for business process modeling. Furthermore, it provides a graphical and easily understandable notation. Petri Nets are object of research for many years and current efforts are focusing on suitable constructs for choreography descriptions. For example, Huangfu et. al. [13] present an approach that addresses the issue of dynamic service composition by modeling service behavior by Object Petri Nets. A service consists of a set of operations and mapping rules from services to Object Petri Nets are introduced. One main drawback of using Petri Nets is that the entire process has to modeled in a single net. In contrast to this, many process algebras directly support parallelism. That allows to model each service separately and then compose them simply by using the parallel operator. Furthermore, Petri Nets do not support all of the workflow patterns [1]. Several extension overcome this issue partly but lead often to a lower expressiveness [14]. Consequently, a higher modeling effort is necessary in order to describe processes.

2.3 Distributed Execution of Business Processes

Process execution engines are usually centralized systems that execute an entire business process and manage all its instances. A comprehensive survey of methods for the distributed execution of business processes can be found in [15].

To achieve scalability, a simplistic approach is to replicate the centralized engine and distribute the process instances among the replicas. However, this approach is not sufficient for an Internet of Services scenario. In the following, we only consider methods that allow a distributed execution even of individual process instances.

The distributed process execution engine described in [16] decomposes a single BPEL process into its individual activities and deploys these activities to any set of execution engines. These activities then communicate over the PADRES publish/subscribe platform. An important aim is to deploy activities close to the data they operate on, thereby minimizing network communication costs. The work reported in [15] is based on a similar approach, which allows the distributed execution of an unmodified BPEL process. Algorithms for the automatic partitioning of processes among the partners participating in their execution are provided. In contrast to that, we do not assume that we

[1] http://www.workflowpatterns.com

already have a complete and consistent process model in the beginning - which would be very unlikely in multi-party Internet of Services scenarios. Instead, we allow for distributed modeling and ensure interaction soundness of the process parts by formal verification methods.

3 Subject-oriented Business Process Management (S-BPM)

Subject-orientation gives special consideration to the actors (subjects) in business processes, beside their actions (predicates), and their results (objects) [3, 4]. It is based on the fact that humans, machines, and software services can be modeled in the same manner. Every one of them receives and delivers information by exchanging messages. Humans, e.g., exchanges emails, office documents, or voice messages.

The company Metasonic [17] provides modeling, validation and execution tools that cover the entire business process lifecycle. Subject-oriented business process management (S-BPM) places the focus on the flow of communication among employees instead of defining a rigid central control flow as in traditional BPM. Subject-oriented BPM (S-BPM) acknowledges that employees themselves drive processes. The advantages of subject-oriented BPM (S-BPM) include greater motivation, savings in time and costs, increased flexibility, improved employee integration and simpler compliance.

The following explanation gives a general overview of the subject-oriented approach. The term subject is used for processes executed by one organization, role, IT-organization etc. These are the acting entities in a business process. They synchronize their activities in order to execute a business process triggered by a corresponding event. In a process description from a subject-oriented view, the focus lies on the involved subjects, which have got a role in the process. That can be people or systems performing the process steps (subjects of a process). The subject-oriented view on a process describes which activities the involved subjects (persons, actors, applications) need to perform at what time. Furthermore, which interactions (sending and receiving of messages) are necessary for the coordination of the respective activities. Each subject defines its own control flow, which coordinates and synchronizes itself with the control flows of other subjects via messages. A subject encloses the interactions and activities executed by a certain involved organizational unit, or person within a considered process. During the execution of a process, a subject sends messages to other subjects, receives messages from other subjects or performs internal activities.

To execute business processes, the Metasonic Business Suite [17] contains the tools jLIVE! and jFLOW!. jLIVE! allows to test processes directly without any coding effort. This tool directly acts on the subjects and does not consider *process embedding* (see Section 5.1). Using an Internet browser, all of the process participants can then immediately test the process together on the basis of roles (subject-related).

jFLOW! allows the actual execution of business processes and takes process embedding into account. It provides a process portal that visualizes an employee's running processes in which he or she plays a role in. The process portal guides the employee step by step through his or her processes. As individual process steps are carried out, the process portal automatically informs other involved employees, who then find their tasks in their individual task lists.

3.1 Process Modeling

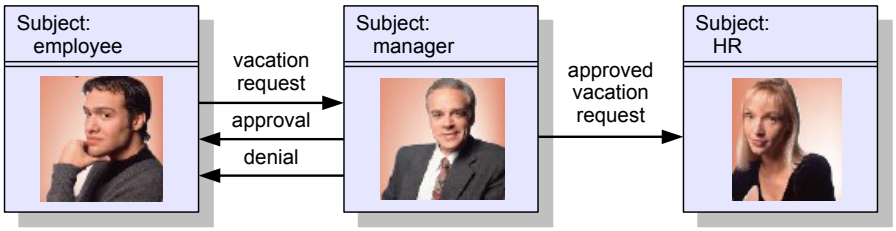


Fig. 1: Subject Interaction Diagram of Vacation Approval Process.

The process model consists of two levels. On the upper level, the involved subjects and the messages they potentially exchange are described in the *Subject Interaction Diagram*. On the lower level, the detailed process is specified for each subject. Figure 1 shows the subject interaction diagram of a vacation approval process. The *Employee* sends a vacation request to his *Manager*. The *Manager* then approves or rejects the request. If it is approved, then the *Manager* also informs the human resources department (*HR*) for bookkeeping.

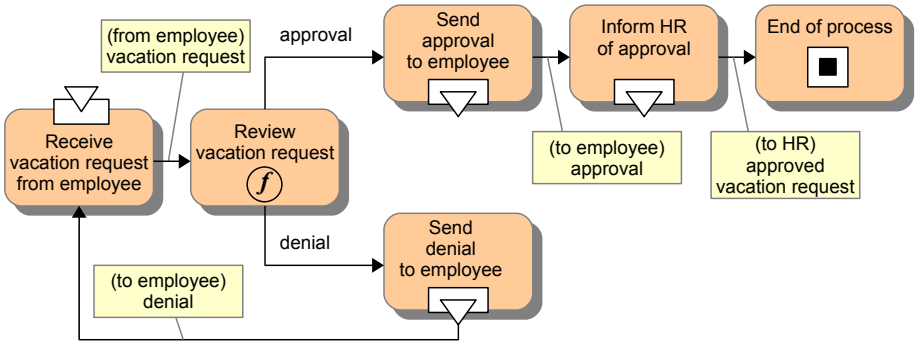


Fig. 2: Process model for subject *Manager*.

Figure 2 shows the internal behavior of subject *Manager*. It is based on the language *Parallel Activities Specification Scheme* (PASS) [5], which is an implementation of subject-oriented modeling. At the basic level, PASS only distinguishes between three basic types of activities: *send message*, *receive message*, and *function*. The *Manager* receives a vacation request from an *Employee*, then approves or rejects it. To demonstrate a loop, the manager waits for a corrected vacation request from the *Employee*, in case he has rejected it in the first place.

4 Formal Description of PASS Processes

PASS is founded on top of the process algebra CCS [18] (Calculus of Communicating Systems) and all language constructs of PASS can be transformed down to pure CCS.

Process algebras provide a suitable means for modeling distributed systems. They offer well-studied algorithms for verification and for determining behavioral equivalences. In addition, the CCS composition operator facilitates a hierarchization and modularization of the model, allowing to handle business processes of arbitrary size.

4.1 Calculus of Communicating Systems (CCS)

Labeled Transition System: Let ACT be a fixed set of actions. A labelled transition system [19] LTS = (PROC, $\rightarrow$) over ACT consists of

- A set PROC of states and
- A set $\rightarrow \subseteq$ PROC$\times$ACT$\times$PROC of transitions between states. Instead of $(s, a, s') \in \rightarrow$ we use the more suggestive notation $s \xrightarrow{a} s'$.

Syntax of CCS: Let $\mathcal{L}$ be a set of labels and let $Act = \{\tau\} \cup \mathcal{L} \cup \{\bar{a} | a \in \mathcal{L}\}$ be the set of all actions. Then the syntax of process P is given by:

$$P ::= 0 \;\Big|\; A \;\Big|\; a.P \;\Big|\; \sum_{i \in I} P_i \;\Big|\; P_1 | P_2 \;\Big|\; P \backslash L \;\Big|\; P[f]$$

with $a \in Act$, $L \subseteq \mathcal{L}$, $f : \mathcal{L} \rightarrow \mathcal{L}$, and $P, P_i \in \mathcal{P}$ where $\mathcal{P}$ is the set of processes. 0 is the inactive process that does nothing.

Semantic of CCS: The semantic of the CCS operators is given by rules of the type:

$$\frac{Pre_i, \ldots, Pre_n}{Imp}$$

The Pre_i are the premises that are met if the implication Imp is satisfied. If $n = 0$ then there are no premises and Imp holds always. This kind of rules are called *Structural Operational Semantic* rules and were introduced by Plotkin [20].

Name	Sym.	Structural Operational Semantic
action	.	$\frac{}{a.P \xrightarrow{a} P}$
process identifier	:=	$\frac{P \xrightarrow{a} P'}{A \xrightarrow{a} P'}\ if\ A := P$
choice	$\sum$	$\frac{P_j \xrightarrow{a} P'_j}{\sum_{i \in I} P_i \xrightarrow{a} P'_j}\ j \in I$
parallel composition	|	$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}\quad \frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'}\quad \frac{P \xrightarrow{a} P',\ Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
restriction	$\backslash$	$\frac{P \xrightarrow{a} P'}{P \backslash L \xrightarrow{a} P' \backslash L}\ a, \bar{a} \notin \mathcal{L}$
renaming	f	$\frac{P \xrightarrow{a} P'}{P[f] \xrightarrow{f(a)} P'[f]}\ where f(\tau) = \tau, f(\bar{a}) = \overline{f(a)}$

4.2 Mapping PASS to CCS

There are three basic symbols to model the internal behavior of subjects in PASS: *send*, *receive* and *function*. Send and receive manage the interaction among different subjects and we denote them as communication activities. They are the equivalent to the send and receive actions of CCS. In the example below, receiving a vacation request is denoted as $vacation\_request$. Send actions are marked with an overline, e.g., sending the approval is denoted as $\overline{approval}$.

The function symbol of PASS is used to describe the call of an internal function, which may be implemented by a software service or it may require a user interaction. Such invokes are modeled by sending and receiving messages to a function library, which is specific for each subject. The following CCS code describes the behavior of subject *Manager*:

$$
\begin{aligned}
Manager_{Interface} &:= vacation\_request.\overline{int\_vacation\_request}.\\
&\quad (int\_approval.Manager^1_{Interface}\\
&\quad + int\_denial.Manager^2_{Interface})\\
Manager^1_{Interface} &:= \overline{approval}.\overline{approved\_vacation\_request}.0\\
Manager^2_{Interface} &:= \overline{denial}.Manager_{Interface}\\
Manager_{Library} &:= int\_vacation\_request.(\overline{int\_approval}.Manager_{Library}\\
&\quad + \overline{int\_denial}.Manager_{Library})\\
Manager &:= Manager_{Interface} \mid Manager_{Library}
\end{aligned}
$$

4.3 Hiding the Internal Behavior

All internal communication can be hidden by applying the restriction operator:

$$Manager := (Manager_{Interface} | Manager_{Library}) \\ \backslash \{int\_vacation\_request, int\_approval, int\_denial\}$$

Inserting the process descriptions of $Manager_{Interface}$ and $Manager_{Library}$ and resolving the internal behavior afterwards, leads to a simpler term. This is shown in Figures 3 and 4 by an example of a simple subject A. A consists of the interface A_1 and the function library A_L. Using the simplification of equation 1, the behavior of subject *Manager* can be rewritten as:

$$Manager := vacation\_request.\tau.(\tau.Manager^1 + \tau.Manager^2)$$
$$Manager^1 := \overline{approval}.\overline{approved\_vacation\_request}.0$$
$$Manager^2 := \overline{denial}.Manager$$

This describes the behavior of subject *Manager* which is visible externally. In an inter-business scenario, this allows the interacting parties to keep the details about their internal processes private.

4.4 Verifying Process Compatibility

The overall vacation approval process is given by combining all subjects using the parallel composition operator:

$$Vacation := (Employee \mid Manager \mid HR)\backslash\{L\}$$
where L denotes the set of all free messages of $Vacation$

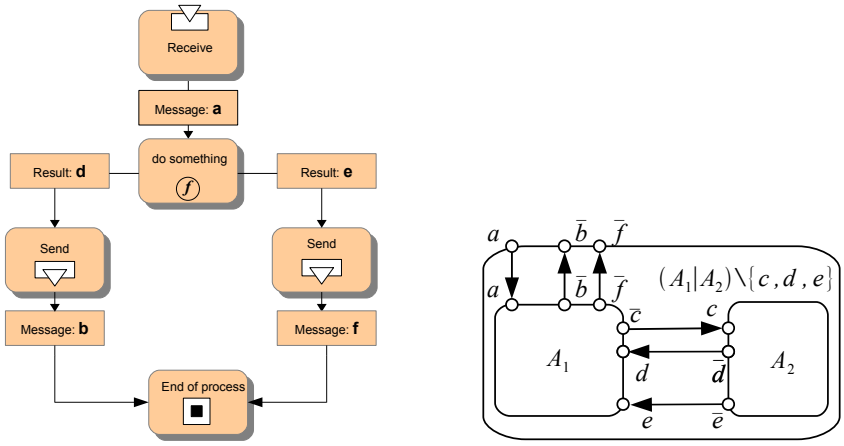
We currently use the CWB-NC Workbench [21] for running verifications on the resulting CCS code. CWB-NC supports various behavioral equivalences as well as model checks. The model checker determines whether systems satisfy formulas written in an expressive temporal logic, the modal μ-calculus.

Firstly, this allows us to perform a choreography conformance check. In a valid composition, it must be ensured that the involved services are able to communicate properly with each other. Secondly, reachability analysis is used to ensure that all end states of the subprocesses can be reached, where applicable.

4.5 Distributed Modeling

In a scenario where multiple different businesses cooperate, the processes for different subjects will typically also be modeled in different places. In addition, the businesses are usually not willing to fully disclose their internal processes. As shown in section 4.3, the internal behavior can be removed from the full process specification using the CCS restriction operator and businesses only have to expose the resulting process description which only describes all external interactions of a subject.

The process compatibility can also be verified in a peer-to-peer fashion. For example, if the Employee wants to verify if his process is compatible with that of the



(a) PASS representation (b) Blackbox representation

Fig. 3: The internal behavior of a subject $A = (A_1|A_2)\backslash\{c, d, e\}$ in two different representations.

$$
\begin{aligned}
A_1 &:= a.\bar{c}.(d.\bar{b}.0 + e.\bar{f}.0) \\
A_2 &:= c.(\bar{d}.A_2 + \bar{e}.A_2) \\
A &:= (A_1 \mid A_2)\backslash\{c, d, e\} \\
&\to (a.\bar{c}.(d.\bar{b}.0 + e.\bar{f}.0) \mid c.(\bar{d}.A_2 + \bar{e}.A_2))\backslash\{c, d, e\} \\
&\to a.(\bar{c}.(d.\bar{b}.0 + e.\bar{f}.0) \mid c.(\bar{d}.A_2 + \bar{e}.A_2))\backslash\{c, d, e\} \\
&\to a.\tau_c.((d.\bar{b}.0 + e.\bar{f}.0) \mid (\bar{d}.A_2 + \bar{e}.A_2))\backslash\{c, d, e\} \\
&\to a.\tau_c.(\tau_d.(\bar{b}.0 \mid A_2) + \tau_e.(\bar{f}.0 \mid A_2))\backslash\{c, d, e\} \\
A' &= a.\tau_c.(\tau_d.\bar{b}.(0 \mid A_2) + \tau_e.\bar{f}.(0 \mid A_2))\backslash\{c, d, e\}
\end{aligned}
$$

$$
\Longrightarrow A' = \; a.\tau_c.(\tau_d.\bar{b}.0 + \tau_e.\bar{f}.0) \tag{1}
$$

Fig. 4: After the reduction of the subject equation of the subject A the process library A_2 is at its initial point again. A further solving of the equation is not possible without interactions to the environment of A.

Manager, then the Manager would only expose the parts of his process that are relevant for the interaction with the Employee. Hence, he would remove all communication with third parties using the restriction operator:

$$Manager_{Emp} := Manager \backslash \{approved\_vacation\_request\}$$

which yields:

$$Manager_{Emp} := vacation\_request.(\overline{approval}.0 + \overline{denial}.Manager_{Emp})$$

As described in section 4.4, the employee can now verify the following expression:

$$Vacation_{Emp} := (Employee \mid Manager_{Emp}) \backslash \{L\}$$

where L denotes the set of all free messages of $Vacation_{Emp}$

5 Process Execution

PASS processes can be directly executed on a suitable execution engine. From the process model to the final execution of a process instance the following two additional steps are necessary: *embedding* and *instantiation*.

5.1 Embedding

First, the process must be embedded into the organization and into the IT. This means that the subjects in the process are mapped to roles or services. If the tasks of a subject are executed by a software service, then a mapping from the subject to the service is defined. If the tasks are fulfilled by a human, then the subject is mapped to a *role*.

For example, the subject *Manager* is handled by the role *Area Head* or *Director* (Figure 5). Which role is the correct one, depends on the context: Employees direct their vacation requests to their manager, which is their Area Head. Area Heads direct their vacation requests to their manager, which is the Director of the lab.

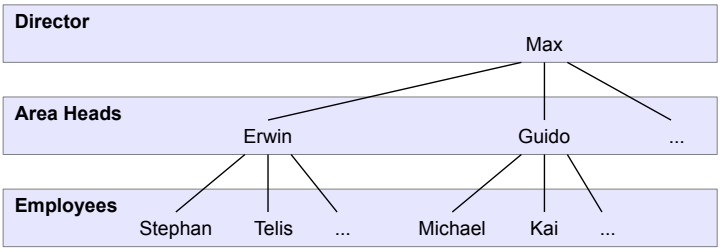


Fig. 5: Organigram of the Telecooperation Lab.

We model the embedding information as a separate aspect apart from the process itself. Compared to that, BPEL does not have a mechanism to handle such embedding. Dynamically choosing between services at runtime must be explicitly coded into the

process. Another approach would be to use proxy services, which decide at runtime to which other service they should forward requests. In contrast to that, the embedding concept reduces the number of process variants that would otherwise have to be modeled explicitly and it does not require additional proxy services.

5.2 Instantiation

Instantiation denotes the creation of a new process instance for a concrete subject carrier. For example, a message is sent to the subject *HR*, which is mapped to the role *Secretary*. *Elke* and *Birgit* have that role and therefore they can accept this message. Now, if *Elke* accepts the message, then a new process instance is created. From this time on, she takes over the subject *HR* / the role *Secretary* for this concrete process instance.

6 Message Routing

When such processes are executed in a distributed system on multiple communicating engines, then the question arises how to correctly route messages from a process instance to remote process instances, while taking embedding information into account. Because this requires a communication abstraction that supports dynamic endpoints and multicast, we use publish/subscribe.

6.1 Publish/Subscribe

A publish/subscribe system consists of a set of clients that asynchronously exchange messages, decoupled by a message broker. *Clients* can be characterized as *producers* or *consumers*. Producers *publish* messages, and consumers *subscribe* for messages by issuing *subscriptions*, which are essentially stateless message filters. Consumers can have multiple active subscriptions, and after a client has issued a subscription, the notification service delivers all future matching notifications that are published by any producer until the client cancels the respective subscription. The message broker is a logically centralized component responsible for distributing messages arriving from multiple publishers to its multiple subscribers. Publish/subscribe has the following characteristics [22]:

- **Space decoupling:** producers do not individually address consumers while publishing messages. Instead, they publish messages through the message broker, and subscribers receive these messages indirectly through the message broker. Publishers do not usually hold references to the consumers and they are not aware of how many consumers are participating in the interaction.
- **Time decoupling:** producers and consumers do not need to actively participate in the interaction at the same time. In particular, a subscription causes messages to be delivered even if producers join after the subscription was issued. In a plain publish/subscribe system, notifications are retained only as long as it takes to distribute them to current subscribers. Some brokers deliver messages to consumers through a queue. This allows consumers to temporarily disconnect from the system. All messages stored while the consumer was offline are dispatched upon reconnect.

The simplest publish/subscribe addressing scheme is based on the notion of *channels* or *topics*. Participants explicitly publish notifications to one or more channels, which are identified by a name (e.g., a string or an URL-like expression). The concept of channels is very similar to the concept of *groups* as defined in *group communication*. Subscribing to a channel can be viewed as becoming a member of the group, and publishing is performed by broadcasting the notification to all members of the group. The part of the notification that is visible to the event service is the identifier of the channel. Since there is no interplay between different channels, each channel can be considered as an event service of its own. A subscription-based Publish/Subscribe system with channel-based addressing supports the following operations:

$\mathbf{sub}(X, C)$	Client X subscribes to channel C
$\mathbf{unsub}(X, C)$	Client X unsubscribes from channel C
$\mathbf{notify}(X, n)$	Client X is notified about n
$\mathbf{pub}(X, n)$	Client X publishes n

Clients register their interest in specific kinds of notifications by issuing subscriptions via the $\mathbf{sub}(X, C)$ operation. From that time on, all notifications matching the channel C are delivered to the client. The client receives notifications through the notify operation, which is an output operation and often implemented as a call-back. Each client can have any number of active subscriptions. A client can revoke subscriptions individually by issuing the $\mathbf{unsub}(X, C)$ operation.

6.2 Communication Between Subjects

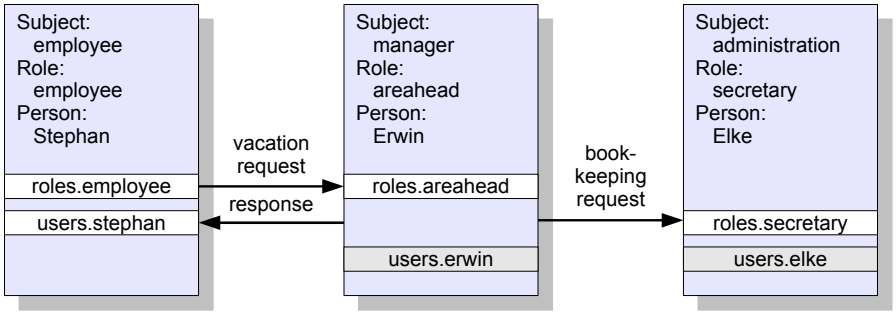


Fig. 6: Communication between subjects using Publish/Subscribe.

Figure 6 shows the publish/subscribe channels involved in the vacation approval process. When Stephan, Erwin, and Elke use separate process execution engines, they issue the following subscriptions:

sub(Stephan, users.stephan)	**sub**(Erwin, users.erwin)	**sub**(Elke, users.elke)
sub(Stephan, roles.employee)	**sub**(Erwin, roles.areahead)	**sub**(Elke, roles.secretary)

Each process engine subscribes two channels from which it can receive messages. The first channel name is constructed from the role name and is used to receive messages when the subject *manager* of a process instance is not bound yet. It is best practice to send messages to roles whenever possible instead of addressing messages directly to specific persons. For example, an employee of area *SE* would send his vacation request to *areahead.se*. This indirection allows to put multiple persons in charge for a role or it allows to temporarily assign the role to another person, e.g., if Erwin takes vacation.

The second channel name is constructed from the name of the person. This allows to send messages to a specific person and is typically used for reply messages. When Erwin accepts or declines the vacation request from Stephan, then Stephan is already bound for subject *employee* in the process instance and it makes only sense to send that reply directly back to Stephan.

6.3 Embedding

Two use cases are particularly interesting with respect to embedding: *context resolution* and *anycast*.

Context Resolution: For example, Stephan issues a vacation request and thereby creates a new process instance. It is clear that he is the subject *employee* in the process. The overall process can only start at this subject, because it is the only subject that does not start with a receive action. The process engine of *Stephan* is configured with all roles he has. Hence, it can be determined from this information that his role is *employee*. Next, he sends the vacation request message to his *manager*.

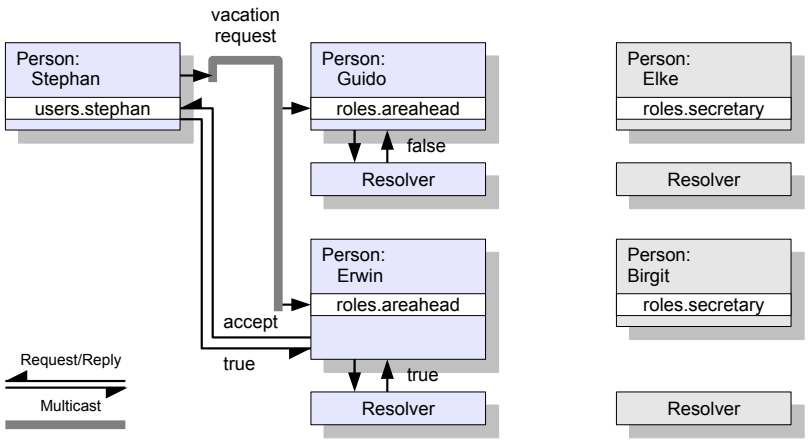


Fig. 7: Routing of a message to a single destination based on context.

The message flow is shown in Figure 7. Sending the message to his manager means that he broadcasts it to all *areaheads*. The process engine of each area head now passes

the received message to the *resolver* service. The resolver service returns a boolean value indicating if the message should be accepted or not. The resolver of Guido returns false, because Guido is not responsible for Stephan. Hence, Guido silently discards the message. The resolver of Erwin returns true, because Stephan is in his area. Erwin now calls *accept* on the engine of Stephan to acknowledge the successful reception of Stephan's request.

Anycast: The first example demonstrated how messages can be routed to a role, by selecting a specific responsible person based on the context. It is also possible that more than one person has the requested role and is able to process the message. This is shown in the anycast scenario in Figure 8. Here, Elke and Birgit both have the role *secretary* and it does not matter if Elke or Birgit handles the bookkeeping request.

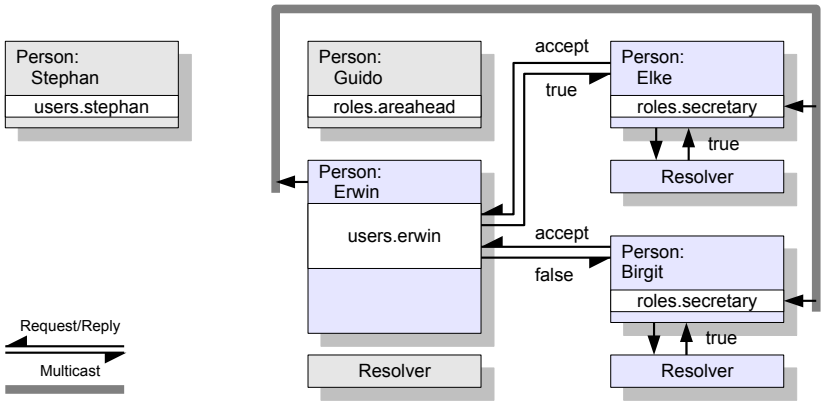


Fig. 8: Routing of a message to multiple destinations having the same role and equal status.

The interaction goes as follows. Erwin sends the bookkeeping request to the channel *roles.secretary* which means that it will be multicasted to both, Elke and Birgit. Elke's engine asks the resolver if the message should be accepted, which returns true. Now, she calls *accept* and Erwin's engine returns *true*. Erwin now has the information that Elke has accepted the message and will continue with the process. If somebody else calls *accept*, then Erwin's engine will always respond with *false*. Birgit's resolver also responds true. Now, when she calls *accept*, then Erwin's engine returns *false* and Birgit's engine silently drops the request.

7 Implementation

In the following, the implementation of our process choreography engine is described.

7.1 MundoCore

We used the communication middleware MundoCore [23] as Publish/Subscribe system. MundoCore is a flexible communication framework that allows to integrate service-oriented systems spanning multiple platforms and programming languages. It supports different communication abstractions, such as Publish/Subscribe and Remote Method Calls, over different transport and invocation protocols. MundoCore provides a common set of APIs for different programming languages (Java, C++, .NET) on a wide range of different devices. With a minimal footprint of approximately 42 KB, MundoCore can be run on a broad spectrum of computing devices, ranging from servers down to mobile phones or sensors. The Publish/Subscribe implementation of MundoCore operates fully peer-to-peer and does not require any centralized components for message routing.

7.2 Process Execution Engine

Our engine executes subject-oriented business process management (S-BPM) models created with the jPASS modeling tool and has a functionality that is similar to jFLOW. Each started engine executes a subset of the subjects defined in the process. Through interconnection of several engines, the whole process can be executed. The engine supports PCs and mobile devices, i.e., Android-based phones.

Figure 9 shows screenshots of the PC and Android versions of the process execution engine. For example, the employee uses the PC version, the manager uses the engine on his mobile phone, and the HR department uses an automatized process.

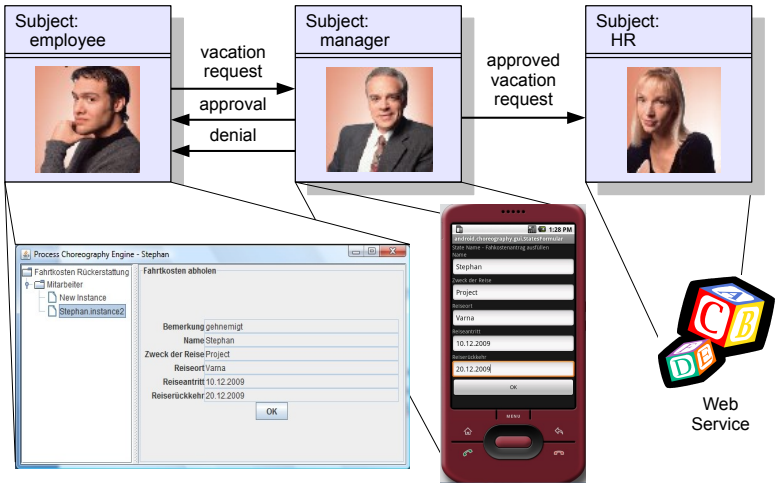


Fig. 9: Screenshots of the process execution engine on PC and Android mobile phone.

8 Conclusion

We have described an approach for the distributed execution of business processes, based on subject-oriented process modeling and a Publish/Subscribe communication middleware. Our system provides several benefits in the following two important use cases:

- A process model can be split up along its subjects and be executed in a distributed manner on several process execution engines. This is suitable for use within an enterprise and allows to execute processes on servers of different departments. In addition, employees can execute processes on their mobile devices and work on processes even when the do not have Internet connectivity.
- If cooperating processes are modeled in a distributed manner, which is often the case when different businesses cooperate, then we can still perform the same model checks as if the overall choreography was modeled as a single, centralized process. In addition, the businesses do not have to disclose their detailed internal processes.

The presented concepts have been implemented in our process choreography engine, which runs on PCs, servers, and Android-based mobile phones.

In our future work we plan to investigate how our approach can be applied to process models of the upcoming BPMN 2.0 standard and how BPMN 2.0 models could be transformed to PASS.

Acknowledgments This work was supported by the Theseus Programme, funded by the German Federal Ministry of Economy and Technology under the promotional reference 01MQ07012.

References

1. BMWi: TEXO – Business Webs in the Internet of Services. http://theseus-programm.de/scenarios/en/texo.html (2009)
2. Puhlmann, F., Weske, M.: Interaction Soundness for Service Orchestrations. In: ICSOC 2006. Volume 4294 of LNCS. (2006)
3. Schmidt, W., Fleischmann, A., Gilbert, O.: Subjektorientiertes Geschäftsprozessmanagement. HMD - Praxis der Wirtschaftsinformatik (266) (April 2009) 52–62
4. Fleischmann, A., Lippe, S., Meyer, N., Stary, C.: Coherent Task Modeling and Execution Based on Subject-Oriented Representations. In: Task Models and Diagrams for User Interface Design (TAMODIA). Volume 5963 of LNCS., Springer (2009) 78–91
5. Fleischmann, A.: Distributed Systems: Software Design and Implementation. Springer (1994)
6. Koning, M., Sun, C., Sinnema, M., Avgeriou, P.: VxBPEL: Supporting Variability for Web Services in BPEL. Information and Software Technology **51**(2) (2009) 258–269
7. Wu, Z., Deng, S., Li, Y., Wu, J.: Computing Compatibility in Dynamic Service Composition. Knowledge and Information Systems **19**(1) (2008) 107–129
8. Bordeaux, L., Salaun, S., Berardi, D., Mecella, M.: When are Two Web Services Compatible. In: Technologies for E-Services. Volume 3324 of Lecture Notes in Computer Science., Springer (2005) 15–28

9. Lapadula, A., Pugliese, R., Tiezzi, F.: A Calculus for Orchestration of Web Services. In: Programming Languages and Systems. Volume 4421 of Lecture Notes in Computer Science., Springer (2007) 33–47
10. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A Calculus for Service Oriented Computing. In: Service-Oriented Computing (ICSOC). Volume 4294 of Lecture Notes in Computer Science., Springer (2006) 327–338
11. Agarwal, S., Rudolph, S., Abecker, A.: Semantic Description of Distributed Business Processes. In: Proceedings of AAAI Spring Symposium – AI Meets Business Rules and Process Management. (2008)
12. Markovic, I., Pereira, A.C., Stojanovic, N.: A Framework for Querying in Business Process Modelling. In: Multikonferenz Wirtschaftsinformatik. (2008) 1703–1714
13. Huangfu, X., Shu, Z., Chen, H., Luo, X.: Research on Dynamic Service Composition Based on Object Petri Net for the Networked Information System. Fifth International Joint Conference on INC, IMS and IDC (2009) 1075–1080
14. Puhlmann, F.: On the Application of a Theory for mobile Systems to business process management. PhD thesis, University of Potsdam, Germany (2007)
15. Wutke, D.: Eine Infrastruktur für die dezentrale Ausführung von BPEL-Prozessen. PhD thesis, Universität Stuttgart (2010)
16. Muthusamy, V., Jacobsen, H.A.: BPM in Cloud Architectures: Business Process Management with SLAs and Events. In: Business Process Management. Volume 6336 of LNCS., Springer (2010) 5–10
17. Metasonic: Welcome to the Future of BPM: S-BPM. http://www.metasonic.de (2010)
18. Milner, R., ed.: Communication and Concurrency. Prentice Hall PTR (1995)
19. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM **19**(7) (1976) 384
20. Plotkin, G.D.: A structural approach to operational semantics (1981)
21. CWB-NC: The Concurrency Workbench of the New Century. http://www.cs.sunysb.edu/∼cwb/ (2000)
22. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe (2003)
23. Aitenbichler, E., Kangasharju, J., Mühlhäuser, M.: MundoCore: A Light-weight Infrastructure for Pervasive Computing. Pervasive and Mobile Computing **3**(4) (August 2007) 332–361 doi:10.1016/j.pmcj.2007.04.002.