Flexible Execution of Distributed Business Processes based on Process Instance Migration

Sonja Zaplata, Kristof Hamann, Kristian Kottke, Winfried Lamersdorf
Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
[zaplata|hamann|3kottke|lamersdorf]@informatik.uni-hamburg.de

Abstract: *Many advanced business applications, collaborations, and virtual organizations are based on distributed business process management. As, in such scenarios, competition, fluctuation and dynamism increase continuously, the distribution and execution of individual process instances should become as flexible as possible in order to allow for an ad-hoc adaptation to changing conditions at runtime. However, most current approaches address process distribution by a fragmentation of processes already at design time. Such a static configuration can be assigned to different process engines near runtime, but can hardly be changed dynamically because distribution logic is weaved into the business process itself.*

A more dynamic segmentation of such distribution can be achieved by process runtime migration even without modifying the business logic of the original process model. Therefore, this contribution presents a migration data meta-model for enhancing such existing processes with the ability for runtime migration. The approach permits the inclusion of intensions and privacy requirements of both process modelers and initiators and supports execution strategies for sequential and parallel execution of processes. The contribution concludes with presenting a conceptual evaluation in which runtime migration has been applied to XPDL and WS-BPEL process instances and, based on these results, a qualitative comparison of migration and fragmentation.

Keywords: Business Process Management, Runtime Migration, Process Fragmentation, WS-BPEL, XPDL

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

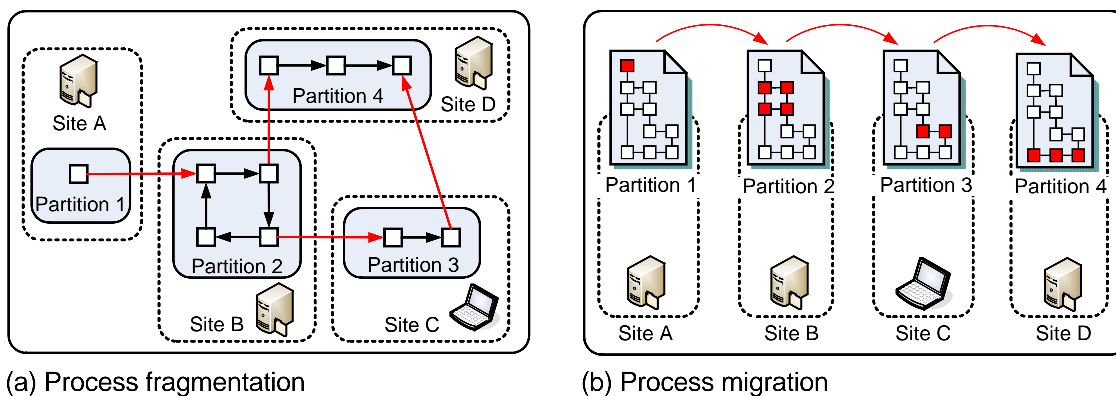
1. Introduction

In today's networked business environments, cross-organizational collaborations composing complementary services and thus realizing new, value-added products gain increasing importance. As a technical representation of such business processes, executable workflows allow for flexible, dynamic and loosely-coupled collaboration among several business partners. The *Business Process Execution Language for Web Services (WS-BPEL)* [16] and the *XML Process Definition Language (XPDL)* [15] are currently two of the most relevant practical approaches. They allow for distributing resources such as employees, machines and services, whereas process control flow logic is typically executed by one single component at one single site [12].

However, due to the autonomy of participants, a single centralized process management system to control the execution of cross-organizational processes is often neither technically nor organizationally desired. As an example, required services and resources often cannot be accessed by a centralized process engine because of technological differences or due to security policies [21]. Furthermore, in some cases the location where a process fragment is executed is relevant to perform the required functionality or is necessary for judicial reasons, e.g. in the context of eGovernment. Related to this, other non-functional aspects such as execution time, performance, navigation cost and capacity utilization can be optimized by load balancing and thus improve flexibility and scalability of participating systems [1]. If e.g. subsequent steps of a process are executed at a remote site, large data transfers can be avoided [12] and potential errors and resulting side effects can be handled more reliably, e.g. in the context of transaction management and compensation of interrelated activity blocks.

Most current research in the area of service oriented architectures is approaching such decentralization of control flow navigation by a *physical fragmentation* of processes – splitting the overall executable process into several subparts which are then distributed to a number of available process engines (cp. Figure 1(a)). In contrast to that, this paper proposes *process instance migration* as a means of *logical fragmentation*, fragmenting only the responsibilities for the execution of the process into a set of sub-responsibilities while preserving the original structure of the process description for all of the participating systems (cp. Figure 1(b)). Such migration is the most "natural" way of executing a distributed process – as inherited by traditional human-based workflow management: A process is described in subsequent steps which are passed from one workplace to another, ensuring the specified task dependencies by sending the tasks to their respective executor when all requisite conditions are satisfied. Logical fragmentation by process migration has several advantages over physical process fragmentation:

- Process migration allows for fragmenting the responsibility to execute a process at runtime – depending on the availability of business partners or other contextual incidences. Furthermore, the granularity of fragmentation and the range of distribution can be selected on the fly by each executing participant.
- Coordination and merging of multiple process fragments is not necessary in the case of sequential execution. Global variables, scopes, errors and transactions are easier to handle, because all these aspects of the process (i.e. data and control flow) are available to all executing parties. Thus, there is less communication and coordination overhead.
- Process migration is applicable to modern distributed systems including mobile devices because it does not depend on a single centralized system and allows for dynamic sharing of restricted resources [13,20].



(a) Process fragmentation (b) Process migration

Fig. 1: Process decentralization variants

However, process migration has also some drawbacks and still includes some interesting challenges which this paper is going to address. First, the process description needs to implement a formal or technical model to communicate the current state of the migrated process instance. To preserve interoperability, this model should not require modifying the original business process [12]. Second, an important motivation for physical process fragmentation is given by the resulting separation of process fragments. If the process is to be fragmented for privacy reasons, process migration lacks proper security mechanisms in order to protect private information carried within the process. Third, if activities within the process should be executed in parallel, process migration alone is not sufficient, but rather process replication is needed in order to split up parallel tasks and allow load-balancing by running them on different machines.

Based on our work in [20], this paper presents an approach to enhance existing processes with *non-intrusive* migration data and an overall system architecture to support runtime process migration among cooperating process execution systems. Therefore, we identify which information has to be attached to the process at design time in order to execute the (logical) process fragments as it was originally intended by the designer or the initiator of the process in whole. Extending previous work, we discuss the distributed execution of parallel process paths and present an initial privacy mechanism to protect the migrating process instance against unwanted changes and unauthorized access. Finally, the approach presented here is realized by a respective prototype implementation which is applied to WS-BPEL and XPDL processes. Results of the evaluation are compared to the general characteristics of physical process fragmentation, before the paper concludes with a short summary.

2. Background and Related Work

Distributed and decentralized process execution becomes increasingly important and, consequently, many such approaches demonstrate the relevance of this research (cp. [8] for a brief overview). A first possible solution for distributing the control flow of a process is to change the service granularity. The activities which should be outsourced are wrapped, encapsulated behind a new service interface and the remaining process model is changed accordingly. A respective approach for WS-BPEL processes is presented by Khalaf and Leymann [9] providing sophisticated concepts to split and distribute specific WS-BPEL elements (e.g. scopes, loops and alternatives). Similarly, the approach of Baresi et al. [2] proposes a distributed service orchestration in WS-BPEL based on partitioning rules and process fragmentation by introducing corresponding *invoke/receive* activity pairs. However, process fragmentation is carried out at design time and is realized by weaving additional activities into the resulting fragments in order to realize a standard-compliant communication between them at runtime.

Another similar approach is to split the original process, deploy the resulting parts at the desired system and induce *choreography* between the separated processes. A choreography-based process management system targeted at dynamic environments is, e.g., represented by *CiAN* [18]. However, choreography and process fragmentation need a joint preparation phase for the physical distribution of each (sub-)process where all participating parties have to be available. Therefore, this approach is more advantageous in case of a similar recurrent execution of the same process than for spontaneous reactions to (infrequent) ad-hoc changes. As also criticized by Martin et al. [12] both solutions imply heavy changes in the original process model and additionally require the introduction and maintenance of new services. Thus, on the one hand, these unnecessary changes to the original process model are not motivated by the original business process, but by infrastructural constraints [12]. In consequence, the authors propose a non-intrusive approach for process fragmentation and decentralized execution. Here, fragmentation is achieved by transforming the orchestration logic represented in WS-BPEL into a set of individual activities which coordinate themselves by passing tokens over shared distributed tuple-spaces. Decentralized process execution has also been considered in Mentor [14] by partitioning a process based on activity and state charts. Addressing more dynamic environments, the approach of *MobiWork* [7] realizes mobile workflows for ad-hoc networks and is focused on the allocation of tasks to mobile participants also using process fragmentation to generate "sub-plans".

However, all presented approaches support at most dynamic allocation and assignment on the basis of a *static fragmentation*. All fragments and responsible parties are determined either at design time or once after invocation but mostly before executing the first activity of the process instance. Considering long-running processes, this flexibility may not be enough in order to also allow reactions to spontaneous contextual changes. In contrast, dynamically *continuable runtime segmentation* implies that fragments and responsible parties are determined dynamically according to the current context and with respect to previous results and requirements of upcoming activities during the actual execution of the process instance.

A way to address such dynamic behavior is based on runtime migration of entire process descriptions. Migrating workflows as a basic concept for process automation have been introduced by Cichocki and Rusinkiewicz [5] in 1997. More recently, the framework *OSIRIS* [17] relies on passing control flow between distributed workflow engines in order to execute service compositions. Process data is kept in a distributed peer-to-peer-database system which can be accessed from each node participating in the process execution. In *Adept Distribution* [3] a similar approach to process fragmentation and decentralized execution is presented which supports dynamic assignment of process parts to so-called *execution servers*. The control of a particular process instance migrates from one execution server to another, and the next participant is dependent on previous activities which are able to change the participant to execute the next partition. Related to this, Atluri et al. [1] present a process partitioning algorithm which creates self-describing subprocesses allowing dynamic routing and assignment.

Process migration has also particularly been applied to the area of mobile process execution, e.g. by Montagut and Molva [13]. Their approach relies on passing control flow between distributed WS-BPEL engines and addresses security on an application level by integrating a public/private process model in order to access applications internal to mobile devices. However, such a solution represents a choreography-like approach which only uses process migration in order to hand-over control flow – and thus also has some of the aforementioned disadvantages. The last example is the *DEMAC* middleware [21] which is able to delegate process execution (in whole or in part) to other stationary or mobile process engines. Its restriction to a proprietary process description language is, however, an obstacle to migrate existing business processes and to integrate standard process engines of external parties.

Complementing existing migration approaches, we propose to avoid modifications of the original business process in order to allow for distribution of process instance anywhere during runtime and thereby enable a higher level of flexibility. Furthermore, most migration approaches give scope for rather too much flexibility, i.e. the process instance is migrated without control or the decision about the next participant is determined by one of the execution systems – but often cannot be influenced by the process modeler or initiator. The following section therefore introduces a more independent migration model which can be applied to existing processes while considering the above-mentioned user-defined requirements for logical process fragmentation.

3. Process Instance Migration and Decentralized Execution

There are at least two ways for enabling a process instance to migrate to other systems at runtime: One is to weave migration data into the existing process model (*intrusive migration data*). This can e.g. be realized by inserting migration activities or migration scopes which determine to invoke other process engines using the remaining process description as an input parameter. Alternative paths or loops can optionally specify the distribution to potential migration partners and handle situations where migration fails. Although such an approach could be realized by using the standard elements of process description languages and is thus compatible to existing systems, it only provides low flexibility as migration activities have to be planned in advance (i.e. at design time). Furthermore, this approach requires the original business process instance to be changed which often results in an unwanted mix of business logic and technical execution logic [12]. Compared to physical process fragmentation, there are thus only few advantages.

The alternative is to apply *non-intrusive migration data*. Technically this can be realized by an additional document holding the migration data or as a non-modifying annotation of the process description. Apart from the advantage that business logic does not have to be modified, non-intrusive process instance migration is possible after each activity and the decision about follow-up process engines can be made dynamically at runtime. The general methodology of such non-intrusive migration is depicted in Figure 2. The development starts with the original modeling of the underlying business process which produces a process model, specified in an executable process description language such as WS-BPEL or XPDL (step 1). Optionally in step 2, this process model can now be enhanced by a *migration data model* which holds all information required for migration. In the following, process model and migration data model are deployed (step 3) and can be instantiated by an application or a user (step 4). If required, parameters are passed to customize the process (i.e. normal invocation parameters) or the migration data. The latter is advantageous if the initiator is allowed to influence non-functional aspects about the way a process is executed (e.g. if the user pays for a higher service quality, the selection of migration partners is influenced accordingly). After that, the resulting process instance is executed following the guidelines of the associated migration data. However, if migration data is omitted or migration is not supported, the unaffected process can still be deployed and executed the usual (centralized) way.

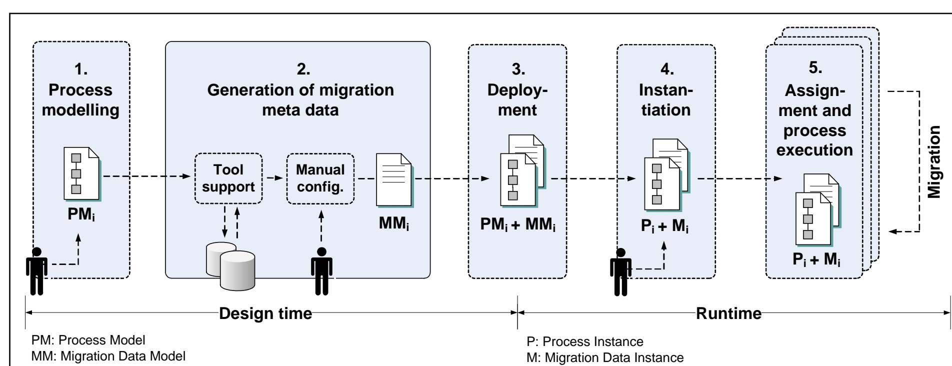


Fig. 2: Process instance migration: methodology

The remainder of this section focuses on the second step of this methodology, i.e. the identification and description of the migration data meta-model (cp. section 3.1) and the necessary enhancements to allow for parallel execution of process parts (cp. section 3.2) and to integrate basic privacy mechanisms (cp. section 3.3). An architecture to deploy and execute the migratable process is outlined in section 3.4.

3.1 Migration Data Meta-Model

The proposed migration data meta-model and its relationship to general process elements are depicted in Figure 3. As a starting point, we assume a common minimal process meta-model consisting of a finite number of *activities* representing the tasks to be fulfilled during process execution, and a finite number of *variables* holding the data which is used by these activities. Activities can represent a specific task *(atomic activities*) or a control flow structure as a container for other activities (*structured activities*). Furthermore, variables can be specified on process level (*global variables*) or at activity level (*local variables*). Optionally, variables can contain an *initial value* which is assigned at design time.

A process description complying with these properties (e.g. XPDL or WS-BPEL) can be enhanced by migration data documenting the execution state of the process (*process state*) and of each activity *(activity state*), such that the progress in processing the activities is well-defined and visible for every participating device at any time during execution. The process state can take a value from the *migratable process lifecycle model* [21] as depicted in the upper right corner of Figure 3. As long as an activity can be executed at the local process engine, there is no need to search for another execution partner to accomplish this task. Consequently, the process is not transferred before all of the currently executed atomic activities are completed which preserves the process's consistency and integrity of its data. Avoiding splitting up such atomic tasks, the safe state *Option* defines a stable point to transfer a process during its execution. In contrast, the process is regarded to be in the state *Running* if atomic activities are in the state *executing*. Other states are used for the administration of the process, e.g. to keep it for logging purposes or to denote an error. The state of each activity is represented by an element of the *activity life cycle state model* based on the established lifecycle model presented by Leymann and Roller [10].

In addition to that, a set of activities can be referenced as *startactivities* to mark the first activity to be executed after process migration. The model allows for multiple startactivities in case the order in which the activities have to be executed is irrelevant or the activities should be processed in parallel. The indication of a start activity requires each activity to have a unique identifier (*ID*) in order to describe a pointer to this activity. Besides the state of the process and its activities, also the states of the variables have to be documented. As process migration is unable to cope with applications which keep part of their state externally, e.g. data stored in an external database, the *current value* has to be copied and attached to the migration data.

Up to this point, basic migration data can be generated automatically, i.e. by setting the process to the state *created* and all activities to *inactive* (cp. first part of step 2 in figure 2). If variables have been specified with an initial value, the given value is set as the current value of the variable. However, the process modeler or the actual initiator often wants to influence the way the distributed process is executed. If the process is going to be migrated, one of the most important questions is where the execution of the upcoming activity should be performed. Furthermore, additional data has to be transported to enable security and traceability of the process. As this could be determined by various (application-dependent) aspects, the following extensible migration model elements can be specified by the process initiator (cp. Figure 3):

The *selection type* determines which strategy is used to assign an activity to a specific process engine. If the selection type is *undefined* (default) the process engine which is currently working on the process instance decides about further migrations. Thereby, it is able to shift processes to other engines which e.g. have access to required resources or which operate at a better performance. In contrast, the type *fixed participant or role* determines that a specific executing entity (e.g. a human or a concrete process engine) or a subject of a defined group of such entities (e.g. a process engine belonging to the role "cooperation partner") has to execute the process or a specified set of activities. More dynamically, as proposed by [3], the next participant can also be picked from a *variable* within the process description itself. If no such entities should be specified, but the participant should be selected as a result of a computation (e.g. picking the process engine which can execute as much of the process as possible), the respective *algorithm* is referenced. Finally, the selection can be based on specific *quality of service or context* requirements such as current workload or geographical location. Associated information about entities, algorithms or non-functional criteria can be included as an additional entry in the migration data or can be referenced (e.g. a URL). Attributes which are attached to process-level apply to all included elements, i.e. activities and variables. However, such attributes can be overwritten by local attributes on activity-level. This allows for specifications such as "all participants should be selected according to the quality-of-service aspect *X*, but the performer of activity *n* must be the fixed participant *P*". Finally, the process modeler can specify which kind of additional data should be collected during process execution, e.g. which participant has actually executed which subset of the process. These requirements and respective collected data can be described in the activity-related *log*.

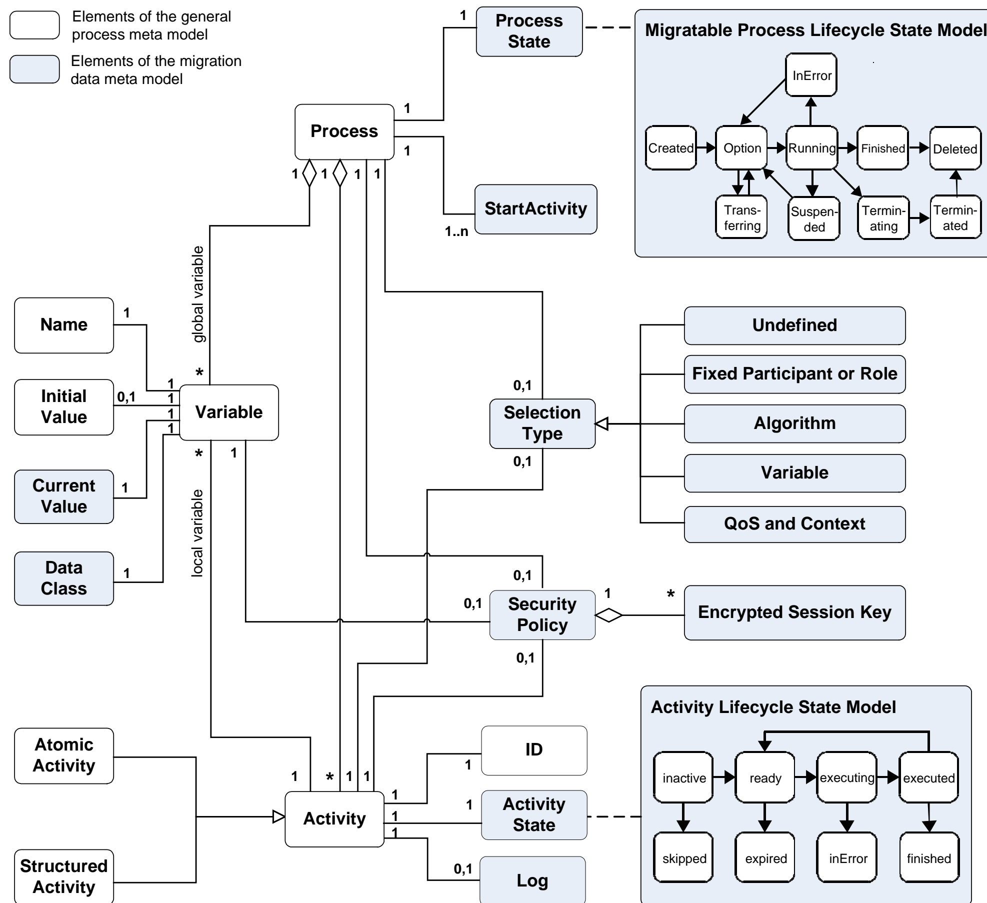


Fig. 3: Migratable process meta-model

Since procedures to allocate and select suitable participants depending on a given set of tasks in decentralized environments have already been established (e.g. [18,20]), the specification of selection algorithms is not part of this paper. Instead, the next subsections focus on the prerequisites for a distributed execution of parallel process paths and the required privacy of critical process parts to establish a basic model for the *security policy* of the presented migration model.

3.2 Distribution of Parallel Process Paths

In case of a sequential execution of the process, the efforts of coordination can often be reduced to a (relatively simple) delegation resp. migration protocol (cp. [21]). As long as the process is transferred to exactly one participant, also the execution of *parallel sections of the process* is uncritical as the states of each included activity are well defined and data dependencies can be handled locally. However, since the execution of parallel paths on a single machine cannot be considered as "real parallelism", a copy of the (entire) process can optionally be distributed to different participants which are each responsible for the execution of one of the parallel paths. In this work, this strategy is referred to as *process replication*.

To distribute a parallel section of the process, the responsible process engine decides to execute an arbitrary or predefined parallel path of the section and thereby sets its first activity to the state *executing*. While in this state, it produces a snapshot of the process description as a copy of its own process and forwards this copy to exactly one other system. Because the path chosen by the first device is already in the state *executing*, upcoming devices can only select one of the remaining parallel paths. Using this strategy, there is always one device responsible for a specific path of the process description and it is therefore also responsible for error handling along this path. In order to synchronize parallel paths, however, there has to be a defined meeting point (e.g. a *fixed participant*) which nevertheless can be chosen at runtime. The participating process engines then pass their copies of the process description to the given participant. A service at the meeting point collects all incoming parallel paths belonging to a shared identifier and merges the copies to a single process description. If required, this one can be forwarded again to continue execution.

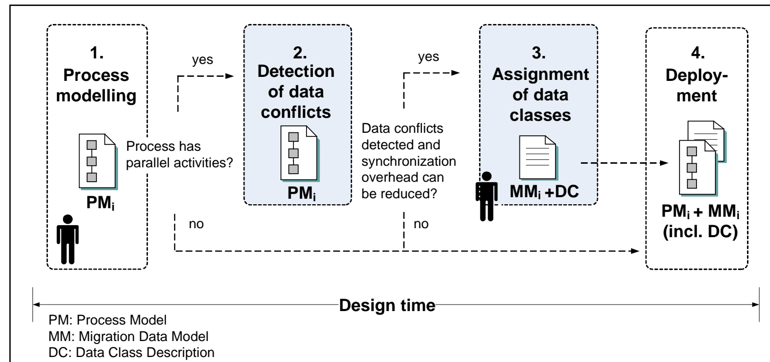


Fig. 4: Preparation for reduction of synchronization overhead

However, if *shared data objects* are used in more than one of these parallel sections, a separate execution could lead to undesired or even wrong results. In consequence, distributed parallel execution of migrating process instances needs advanced coordination mechanisms for synchronization. If the effort for synchronizing variables on parallel paths is considered to be critical (e.g. in mobile environments where communication networks have a low bandwidth) the process designer can try to reduce synchronization overhead already at design time. The according procedure is shown in Figure 4: The process model is checked for process variables which are read and/or written in more than one parallel path. In case a *data conflict* is detected (i.e. coordination and/or synchronization of process variables may be necessary in order to ensure a correct execution), the process modeler can solve unintended data dependencies or assign so-called *data classes* which specify under which circumstances the variables have to be synchronized. Inspired from Yu and Vahdat [19], data classes specify application-specific guarantees concerning the consistence of the used data and differ in the method to deal with dependency conflicts. In consequence, the process designer can select the most suitable data class for potentially conflicting process variables in order to further reduce the need for runtime coordination.

As an example, in order to lead to *serializability* as the correctness criterion for process data flow, the data class *serialized* is applied. Accordingly, every dependency conflict which results from variables of this data class has to be resolved by concurrency control, i.e. the affected variables have to be synchronized. *Serialized* is also the default value for ensuring correct process execution if data class selection is omitted. In contrast, the data class *unsynchronized* does not take care of any dependency conflict derived by variables of this class. As a tribute to such a loss of serializability, lost updates can appear. Therefore, the use of this data class is only acceptable under certain application-specific circumstances.

In general, also many other data classes are possible. In some cases, e.g., serializability can be omitted but the accessed data must not exceed a certain *age*. If, for example, a weather service continuously updates the forecast for tomorrow, the particular updates do not essentially differ. The process execution system can realize this by checking the respective variables for changes on other replicas if necessary, i.e. if the process contains a parallel write on this variable and a given period of time has passed until the last check.

Considering runtime, the process is executed the usual way until process execution reaches a branch which results in parallel execution (i.e. an *AND Split*). In this case, the process instance description including all migration data is replicated and distributed to suitable process engines which are now each responsible for the execution of one of the paths (cp. Figure 5). Together with the information extracted from the *data classes*, an algorithm for optimistic runtime conflict resolution is applied in order to reduce coordination overhead among replicates. Final synchronization and merging of replicates can be induced by specifying a *fixed participant* for the execution of the respective *AND Join* activity or by a distributed incremental procedure which allows for determining the synchronization point at runtime. However, this increased flexibility for synchronization also implies a slightly increased coordination overhead. Alternatively, the parallel section of the process can still be executed on a single centralized system. Using general migration data and data class descriptions, this decision can be made at runtime flexibly.

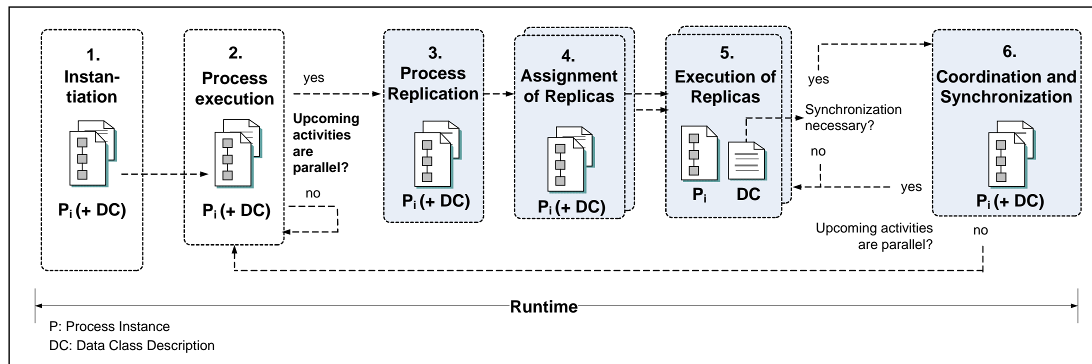


Fig. 5: Execution and synchronization of replicated parallel process paths

3.3 Privacy and Security Considerations

During decentralized execution of a process, its entire information is generally public to subjects which potentially belong to foreign organizations. This may not be acceptable, because the process description often contains private data (e.g. credit card information), private control flow information (e.g. existence of customer complaints), or identities of persons and companies which must not be revealed to or modified by other (external) parties. As another security risk, malicious participants could try to modify parts of the process or the migration data. To prevent such privacy and security threats, the access to process data can be restricted to specified subjects or roles, as e.g. determined in the above mentioned selection types *fixed participant or role*.

Figure 6 shows the general idea of "masking" critical parts of a process description in order to ensure that only dedicated participants can execute sensitive activities and access corresponding data. The approach assumes that potential business partners can communicate with each other without being eavesdropped. Thus, a basic cryptographic key infrastructure is required, such as *PKI* (*Public Key Infrastructure*) or subject-related shared keys. However, encryption of the actual process is more complex, primarily because most process description languages (such as also WS-BPEL or XPDL) allow for the definition of global variables which can be referenced in several activities – and thus might belong to more than one participant. In consequence, these parts cannot be directly encrypted with the personal key of the authorized subjects. Alternatively, the encryption of the different parts of the process (i.e. activities, variables or even the whole process) uses different *session keys* which are only used once. A corresponding *security policy* of such an element therefore contains a number of symmetric keys (e.g. pk_1 and pk_2 in Figure 6). The procedure of key distribution is based on a concept which is derived from broadcast encryption [11] where the same encrypted content is sent to all receiving parties without the need for two-way authentication or authorization. In the approach presented here, the keys necessary for decryption are sent together with the protected content. These keys prevent unauthorized access to the content, but are also themselves protected by cryptography. In case of an existing PKI the entries are encrypted with the public key pub_i of the appropriate subject (cp. Figure 6) and can be unlocked with the private key prv_i. Accordingly, an entry for each authorized subject is created and added to the migration data of the protected process element (cp. Figure 3). As the result of this step, only the legitimate receiver is able to obtain the keys and decrypt the content and even encrypted global variables can be accessed by different authorized subjects using the same session key [4]. Neither an additionally interaction between the process initiator and the subjects nor an authentication is needed. As a positive side-effect, the use of unique session keys also increases the resistance of the cryptographic approach to attacks.

To additionally ensure the integrity of the process description, the process initiator is optionally able to generate a *MAC (Message Authentication Code)* for each security-related process part. Each process engine owning the appropriate process key pk_i is thus also able to verify the integrity of this part. However, after a participant has modified a part of the process it has to generate a new MAC which confirms the integrity of this part. This possibility is indispensable because variables have to be changed by the subjects during process execution. In addition to the MACs, the process initiator can secure both the existence and the correct sequence of the process parts by a digital signature. In case of an existing PKI each subject can verify the correctness of the signature on the basis of the initiator's certificate, preventing e.g. a later modification of the process sequence. To also prevent replay attacks, an additional timestamp can be added to the signature. The integrity of the process description can be

ensured by storing the digital signatures and the MACs within the migration document which finally has to be secured in a similar way as the process itself.

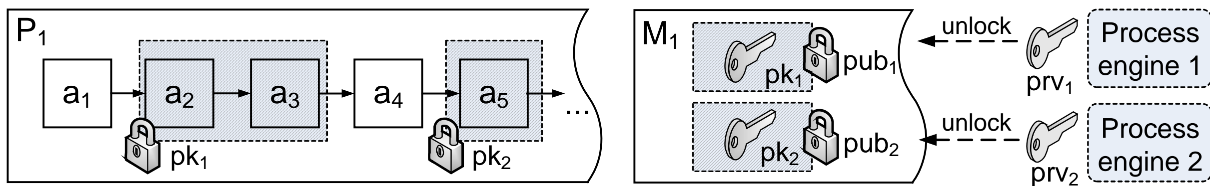


Fig. 6: Process encryption and key distribution

3.4 Execution

The architecture of a corresponding prototype execution support is depicted in Figure 7. Considering the first layer, all potential participants have to provide a compliant interface in order to receive process descriptions from preceding process engines, e.g. represented by a WSDL description. By encapsulating the existing platform and exposing its functionality of cooperative process execution "as a service", the concept of process migration can be embedded into existing system infrastructures. Thus, the interface can be realized by using e.g. a standard web service which receives the process description (P_i) optionally supplemented with migration data (M_i) as an input parameter and returns the identifier of the process and the performer's signature in order to acknowledge its receipt. This service can furthermore be published at a public registry, so it can be discovered and invoked dynamically whenever a migratable process is initiated.

If security mechanisms have been applied, a simple *privacy manager* is responsible for decrypting and encrypting the process and relevant parts of the migration data (layer 2). Encryption of protected process parts can be realized by common procedures such as *AES (Advanced Encryption Standard).* In the case of WS-BPEL which is described in XML syntax, the specifications *Xml Encryption* and *Xml Signature* by the W3C can be utilized to tag encrypted parts and ensure integrity of the migrating process description. However, concerning the "masking" of processes, it has been found that encrypted parts are often causing errors during process execution because the process engine tries to interpret encrypted variables and activities but does not find expected content, e.g. encrypted variables do not match the expected data type. Thus, the privacy manager is also responsible for exchanging non-assigned encrypted parts by temporary dummy variables or activities. As encrypted process parts are not required to actually execute the assigned parts as defined by the security policy, this does not influence process execution at the local site.

The *migration manager* interprets the migration data as specified in section 3.1. It is responsible for passing the given process to the process engine, to update process states, activity states and log files subsequent to execution, and, if necessary, to determine the next process participant according to the given selection type specified for the upcoming activity – potentially making use of existing selection algorithms (layer 3). Considering the integration of the prototype system, it is desirable to completely avoid modifications on existing process execution systems. However, it shows that the underlying process engine has to implement an additional interface for receiving management instructions from the migration manager and for generating events in case of state changes. As most modern process engines already implement a general management interface (such as e.g. the *ActiveBPEL*[1] or *Apache ODE*[2] Management API), migration manager and process engine can be sufficiently decoupled and the modification effort can be limited to a respective adapter component.

[1] http://www.activevos.com/community-open-source.php
[2] http://ode.apache.org/

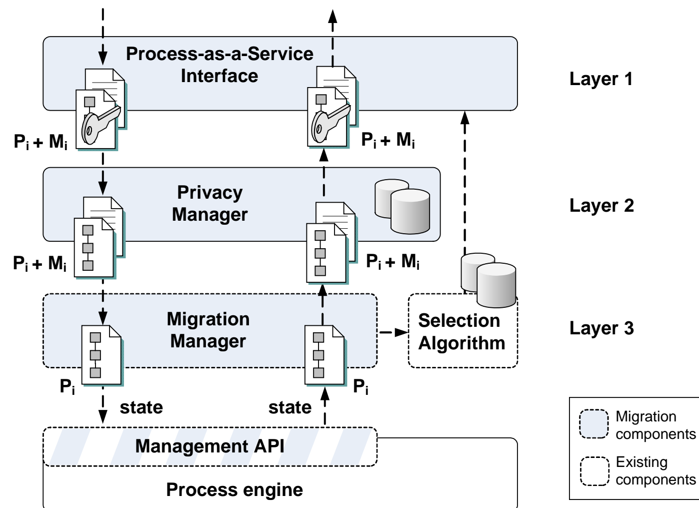


Fig. 7: Runtime migration support

4. Evaluation

So far, a prototype implementation as sketched above has been applied to two existing process management systems: first to the DEMAC [21] process engine which executes XPDL 1.0 processes and, second, to the Sliver [6] process engine which executes a subset of WS-BPEL 2.0 processes. Both process engines can be applied also for *mobile process management* and had to be modified in order to implement the proposed management API. The following discussion shows the most important observations and results also in comparison to physical process fragmentation.

4.1 Runtime Migration of XPDL Process Instances

The *XML Process Definition Language (XPDL)* is a meta-model language developed by the *Workflow Management Coalition (WfMC)* [15] in order to provide an abstract interchange format for different workflow engines. The language is graph-based and allows for the invocation of arbitrary software applications, machines and human resources as atomic *activities*. In order to build more complex control flow structures, activities can be connected by *transitions* and composed to reusable *activity sets*. Branches and parallel execution can be specified by transition restrictions using an AND, OR or XOR *split* activity and control flow synchronization using a *join* activity respectively. *Data fields*, *data types* and *participants* are specified as global parameters of a so called *package* which can hold several *workflows*. However, in order to apply to the common minimal process meta-model and thus to migrate an individual XPDL process instance, it is advantageous if the package contains only one single process description. Thus, the process to be migrated can optionally be cut out of its container and required global parameters are replicated. This does not modify the original description of the relevant process and is uncritical as long there are no inter-process data dependencies. The remaining XPDL constructs have been assigned to the elements of the general model in figure 3. Table 1 summarizes the result of the analysis and shows a conceptual comparison to the general concept of physical process fragmentation.

Atomic XPDL 1.0 activities imply a *one-way* or an atomic *request-response* invocation of resources. Therefore, the activities are, in general, independent from each other and migration can be initiated before or after the execution of an activity, but not within the activity itself. The same is true for the *activity set* as it only replaces an atomic activity with a complex control flow. Thus the atomic activity can simply be set to the state *executing* and the *activity set* can be executed as a regular part of the process.

Composite sequential control flow structures, such as branches or loops, are no explicit elements in XPDL 1.0, but are determined by the developed overall graph structure. Associated conditions are dependent on variable values which are transported with the process. For example, a condition can hold an expression which states that a loop of activities should be executed as long as variable *x=true,* where *x* is a data field of the process and its value is included in the migration data and is thus available to each processing party at every time. Consequently, migration is even possible within iterations – representing another advantage over physical process fragmentation where e.g. loops often have to be distributed as a whole. If the condition has to be evaluated only once (such as in the case of a branch condition) the selected branch is determined by the process's startactivity. In case of process fragmentation, however, fragments and responsible parties are often determined at design-time or at invocation-time. If

a process's transition condition restricting access to parallel or exclusive paths is evaluated at runtime, some of the (physical) process fragments and their respective assignments of executors may never be used. Thus, process migration is often more efficient because it allows integrating the current state of variables at runtime in order to make its assignments. Related to this, the execution of a necessary *dead path elimination* [10] requires further coordination if process fragments are distributed physically. In case of process migration, the dead path can be processed automatically by setting all upcoming activities (until the next join condition) to the *skipped* state. As this information is hold in the migration document, this does generally not involve an extra communication with other participants.

As long as there is only one process path to be executed (XOR *split*), no synchronizations of control flow are necessary and both the split activity and also the execution of following *join* activities can be assigned in a flexible way. Nevertheless, to also support the execution of parallel process paths on different systems, there has to be a defined meeting point for control and data flow synchronization (cp. Section 3.2). Therefore, the performer of the *join* activity must be specified either by the selection type *fixed participant* or must be result of a deterministic *algorithm*. However, such restriction of flexibility is also required for physical process fragmentation. In addition, distributed parallel execution needs advanced coordination mechanisms for both migration and fragmentation. However, using replication instead of fragmentation allows for a local detection of shared variables and thus avoids unnecessary synchronizations.

As an additional remark, there is no explicit *transaction* or *exception* semantics in XPDL 1.0. If activities are, however, part of a transaction (as e.g. possible in XPDL 2.0) migration can optionally be inhibited for the transactional part, so changes are only propagated to other devices after the transaction has committed, or, if activities are *compensable* (which is the default in XPDL 2.0), compensation activities can be executed by other process engines as well, or the process description can be returned to a previous system in case compensation is only possible there.

	Tested XPDL elements	**Process migration**	**Process fragmentation**
Atomic activities	*activity*	possible	possible
Structured activities	*activity set* branches (*XOR*) loops branches (*AND*)	possible possible possible replication and synchronization required	possible possible (transfer of decision) coordination required coordination and synchronization required
Other elements	*transaction (XPDL 2.0)*	for *undo*: to be avoided for *compensation*: possible	coordination required
Dead path elimination	-	automatically	coordination required
Privacy of process parts	-	artificial	automatically
Splitting atomic activities	-	forbidden	no known approach
Data replication	-	only for parallel execution (entire process)	always: *data fields, data types, applications, participants*
Design time distribution	-	possible (assign all activities in advance)	possible
Runtime distribution	-	during execution	once after invocation

Tab. 1: Migratable XPDL processes and comparison to process fragmentation

4.2 Runtime Migration of WS-BPEL Process Instances

WS-BPEL is a block-structured XML-based process description language which allows composing web services. According to the WS-BPEL 2.0 specification by OASIS [16] it is essentially comprised of two kinds of activities: Basic activities for web service interaction (*invoke*, *receive*, *reply*), basic control flow activities (*empty*, *wait*, *exit*, *throw*, *rethrow*) and activities for data manipulation (*assign*). Structured activities are used to compose the basic activities and define control flow dependencies between them (*sequence*, *if then else*, *pick*, *flow*, *while*, *repeat until*, *for each*). Based on this characterization, the activities have been assigned to the elements of the general model in figure 3. Table 2 shows the result of the analysis which was performed in order to evaluate to which extent WS-BPEL processes can be

migrated at runtime. Furthermore, the table shows a comparison to physical process fragmentation and summarizes the following discussion on advantages and disadvantages of both approaches.

Considering atomic activities, it shows that WS-BPEL has a very interactive character which makes the distribution of the control flow logic (both for migration and for fragmentation) more difficult. The *invoke* activity initiates the invocation of a web service which is specified within the process description (or references associated parts such as WSDL files) in either an abstract or a specific way. Thus, migration of a process containing an unprocessed *invoke* activity is not only possible, but even advantageous if the required service is not reachable from the current system. In case of a synchronous service call (*request-response* pattern) the receipt of the response message is part of the atomic activity. In case of asynchronous messaging, sending an associated reply subsequent to a migration is also not critical as the required information about the receiver (e.g. its physical address) can be logged. Nevertheless, receiving a reply (*receive*) requires the specification of a specific participant because the sender of the reply has to know where to send the message. Thus, flexibility of arbitrary distribution is – in this case – limited both for migration and for physical process fragmentation.

The assignment of a variable (*assign*) is not a problem as the current value is stored within the migration data. The same is true for *wait*, *empty* and *exit* activities as these have a rather simple behavior. Notifications about faults are also uncritical as in case of process migration all the relevant information for fault handling (i.e. *scopes, fault handler, compensation handler*) are available to each executing party. If required, the occurrence of faults can also be documented in the log, e.g. if the control flow logic has to return to the failed activity after fault handling is finished. Considering process fragmentation, other process fragments may have to be notified in case of a fault, resulting in an increased coordination overhead.

As indicated above, migration must not happen while an atomic activity is currently executed. However this does not apply for structured activities which only act as a container for other activities. As a consequence, structured activities such as *sequence, if then else* or *while* do not have to be finished in order to allow the migration of the process instance (cp. explanation for branches and loops of XPDL in Section 4.1).

	Tested WS-BPEL elements	**Process migration**	**Process fragmentation**
Atomic activities	*invoke* *reply* *receive* *assign* *wait, empty, exit* *throw, rethrow*	possible possible (log) fixed participant possible possible possible (log)	possible coordination required fixed participant possible possible coordination required
Structured activities	*sequence* *if then else* *while, repeat until, for each* *pick* *flow*	possible possible possible possible, but small risk of missing events coordination required	possible unnecessary fragments coordination required potential replication of events and/or additional coordination coordination required
Other elements	*scope* *fault handler* *compensation handler*	generally available generally available generally available	coordination required coordination required coordination required
Dead path elimination	-	automatically	coordination required
Privacy of process parts	-	artificial	automatically
Splitting atomic activities	-	forbidden	no known approach
Data replication	-	only for parallel execution (entire process)	always: *variables, scopes,* optionally: *events*
Design time distribution	-	possible (assign all activities in advance)	possible (equivalent to service choreography)
Runtime distribution	-	during execution	once after invocation

Tab. 2: Migratable WS-BPEL processes and comparison to process fragmentation

The *pick* activity waits for the occurrence of an event from a set of events and then executes the activity associated with that event. If the process is fragmented physically, this is a problematic issue. Either all the necessary data has to be replicated (i.e. all event/reaction pairs) or the events have to be fragmented as well. If the reaction to an event affects other fragments, additional coordination is necessary. In case of process migration, this is not a problem as the whole spectrum of possible events and reactions is available to the responsible participant. If, furthermore, other activities are temporarily suspended because of the event, the activity states indicate where the execution must be continued. However, the source which is emitting the event has to know where to send the respective messages. Thus, each process participant has to subscribe to each required event as long as it is responsible for the execution of the process instance. During migration time, there is, consequently, a remaining risk that some events may not be noticed.

The *flow* activity contains activities which should be processed in parallel. As long as the process is migrated to exactly one participant, migration within the execution of a flow is uncritical as the states of each included activity are well-defined. Nevertheless, the process cannot be transferred until all atomic activities have reached a stable state and thus may have to wait for long-running activities to be finished. If copies of the process are distributed to other participants to be responsible for the execution of one of the parallel paths, there also has to be a defined meeting point in order to synchronize parallel paths (cp. explanation for XPDL in Section 4.1).

Other interesting aspects discussed in Tables 1 and 2 include privacy of process parts, specification of fixed participants and distribution flexibility. As a drawback for process migration, privacy can only be realized by artificially masking private process parts as proposed in section 3.2, whereas physical fragmentation of the process makes such mechanisms unnecessary. In consequence, the effort for developing migratable processes containing private parts is a little higher. Nevertheless, process migration allows for more flexibility in selecting the most suitable process engine at runtime while still allowing for respecting the interests of the process designer by determining specific participants or selection algorithms. Thus, especially long-running distributed process instances benefit from the possibility to adapt the execution of control flow to changing conditions.

5. Conclusion and Future Work

This paper focuses on distributed process execution involving multiple engines in order to increase flexibility and to improve reactions to ad-hoc context changes. As an alternative to physical process fragmentation, a concept for realizing logical process fragmentation on the basis of process migration has been presented. Compared to physical fragmentation, process migration provides more flexibility by allowing for the distribution of running process instances at runtime while respecting the guidelines of the process modeler. On the other hand, privacy and security-related issues have to be considered explicitly as also addressed in this paper.

Future work includes the evaluation of other practically-relevant process description languages and the implementation of respective migration managers. A prototype system covering the proposed system architecture for XPDL and WS-BPEL processes has already been developed and shows basic applicability of the proposed concepts. Considering privacy support, WS-BPEL process designers must still be careful not to mask multi-level scopes when these are also relevant for public process parts. Based on such requirements, a tool to support process modelers when applying security mechanisms would be useful to facilitate the development of migration data and help process modelers to avoid unnecessary errors.

References

[1] Atluri, V., et al.: *A Decentralized Execution Model for Inter-organizational Workflows*. Distributed Parallel Databases 1/2007, p. 55–83

[2] Baresi, L., Maurino, A., Modafferi, S.: *Towards Distributed BPEL Orchestrations*. ECEASST 3/2006, p.1-14

[3] Bauer, T., Dadam, P.: *Efficient Distributed Workflow Management Based on Variable Server Assignments*. CAiSE 2000, p. 94–109

[4] Bertino, E., Castano, S., Ferrari, E.: *Securing XML documents with Author-X*. IEEE Internet Computing 3/2001, p. 21–31

[5] Cichocki, A., Rusinkiewicz, M.: *Migrating Workflows*. Advances in Workflow Management Systems and Interoperability, NATO 1997, p. 311–326

[6] Hackmann, G., Haitjema M., Gill, C.D., Roman, G.C.*: Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices*. International Conference on Service-Oriented Computing (ICSOC), 2006, pages 503–508.

[7] Hackmann, G., Sen, R., Haitjema, M., Roman, G.C., Gill, C.D.: *MobiWork: Mobile Workflow for MANETs.* Technical Report, Washington University, 2006.

[8] Jablonski, S., et al.: *A Comprehensive Investigation of Distribution in the Context of Workflow Management*. ICPADS 2001, p. 187–192

[9] Khalaf, R., Leymann, F.: *A Role-based Decomposition of Business Processes using BPEL*. IEEE International Conference on Web Services, 2006, p. 770–780

[10] Leymann, F., Roller, D.: *Production Workflow: Concepts and Techniques.* PTR Prentice Hall, 2000

[11] Lotspiech, J., Nusser, S., Pestoni, F.: *Broadcast Encryption's Bright Future*. Computer 8/2002, p. 57–63

[12] Martin, D.,Wutke, D., Leymann, F.: *A Novel Approach to Decentralized Workflow Enactment*. Enterprise Distributed Object Computing, 2008, p. 127–136

[13] Montagut, F., Molva, R.: *Enabling Pervasive Execution of Workflows*. Collaborative Computing: Networking, Applications and Worksharing, 2005, pp. 10

[14] Muth, P., et al.: *From centralized workflow specification to distributed workflow execution*. J. Intell. Inf. Syst. 2/1998) p. 159–184

[15] Norin, R., Marin, M.: *XML Process Definition Language*. Specification WFMC TC-1025, Workflow Management Coalition, 2002

[16] OASIS: *Web Services Business Process Execution Language Version 2.0*. Technical Report, OASIS, 2007

[17] Schuler, C., Weber, R., Schuldt, H., Schek, H.J.: *Scalable Peer-to-Peer Process Management - The OSIRIS Approach*. ICWS 2004, p. 26–34

[18] Sen, R., Roman, G.C., Gill, C.D.: *CiAN: A Workflow Engine for MANETs*. COORDINATION 2008, p. 280–295

[19] Yu, H., Vahdat, A.: *Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services*. ACM Trans. Comput. Syst., 3/2002, p. 239–282

[20] Zaplata, S., Kottke, K., Meiners, M., Lamersdorf, W.: *Towards Runtime Migration of WS-BPEL Processes*. Fifth International Workshop on Engineering Service-Oriented Applications (WESOA'09), 2009

[21] Zaplata, S., Kunze, C.P., Lamersdorf, W.: *Context-based Cooperation in Mobile Business Environments: Managing the Distributed Execution of Mobile Processes*. Business and Information Systems Engineering (BISE) 4/2009, p. 301-314