Process Evolution in a Distributed Process Execution Environment

Pieter Hens [a,*]

Monique Snoeck [a]

Geert Poels [b]

Manu De Backer [a,b,c]

[a] *Department of Decision Sciences and Information Management, KULeuven, Naamsestraat 69, B-3000 Leuven, Belgium*

[b] *Department of Management Information and Operations Management, Universiteit Gent, Tweekerkenstraat 2, B-9000 Ghent, Belgium*

[c] *Department of Management Information Systems, Universiteit Antwerpen, Prinsstraat 13, B-2000 Antwerp, Belgium*

** Corresponding Author*

Abstract

To allow the distribution of control and visibility of cross-organizational process models and to increase availability and performance of the processes, a process model can be fragmented into logically different parts and distributed in the enterprise architecture. Fragmentation algorithms and execution environments which connect the fragmented process model parts together, recreating the original process execution semantics, have been proposed in earlier work. However, a critical challenge that is left open is the ability of the distributed process execution environment to respond effectively to process changes. In this paper, we describe the difficulties, advantages and issues of process model change support in a fragmented and distributed environment. Moreover, we propose a system which tackles the identified issues and allows the propagation and coordination of process changes at runtime in the distributed process execution architecture.

Keywords: Business Process Enactment, Process Evolution, Workflow Change, Distributed Business Processes, Service Oriented Architecture

Process Evolution in a Distributed Process Execution Environment

Business process model fragmentation is the process of splitting a process model that was modeled as a whole into logically different, smaller model fragments with the intention to distribute the fragments over different execution and controlling partners. There are several reasons for process model fragmentation: distribution of ownership and/or coordination across process model fragments; elimination of a single point of failure during process model execution; and increasing availability and performance of the process model execution.

Process model fragmentation allows for the distribution of control and responsibility of the process model. In contrary, using a central execution scheme to operate the complete process flow implies that the responsibility of the entire process execution lies with one organizational entity. However, it is not uncommon that processes are cross-departmental or even cross-organizational, where it is not viable that one single entity has full control over the entire process flow, or even has visibility of the entire process flow. Also, the process may be designed centrally as one unit, but off-shoring and outsourcing process capabilities may require the fragmentation of this process model.

Besides such organizational reasons, executable business process models, i.e. a process model described in an executable process language like BPEL (OASIS, 2007) or YAWL (van der Aalst & ter Hofstede, 2005) may also have technical reasons for their distribution and fragmentation. When the model is executed as one unit by one process engine (*centralized process execution*) and at high loads (i.e. increasing client requests), the engine has to handle a significant amount of process instances simultaneously. For complex processes this requires handling a vast state space, performing complex data transformations and invoking multiple component services (e.g. web services and task managers). This puts a high pressure on the central process engine and performance degrades as the number of process instances increases (Chafle, Chandra, Mann, & Nanda, 2004). Alongside *degradation of the performance*, centralized execution also adds a *single point of failure* to the process execution architecture. Services (e.g. web services) are distributed and decentralized, but the decision logic and coordination (composition) of these services is still located at one point (i.e. the process engine). Failure of the coordinator means failure of the entire process, even if the services themselves are still available and ready to be executed (Chafle et al., 2004; Muth, Wodtke, Weissenfels, Dittrich, & Weikum, 1998). Process model fragmentation and distributed execution addresses these issues.

Many different techniques for process model fragmentation have been proposed in literature. For example, Chafle et al. (2004) use program dependency graphs, a tool borrowed from compiler optimization, to split up the process flow. Their goal is to reduce the network traffic involved. For the same reasons, Fdhila et al. (2009) fragment the process flow using dependency tables. To increase availability and failure-resilience Muth et al. (1998) perform process model fragmentation using state and activity charts and Khalaf et al. (2008) fragment a BPEL flow according to predefined swim-lanes to enable distribution of ownership and coordination. The proposed techniques differ in the way and reasons processes are fragmented, but the result is, however, always the same: a set of logically different fragments distilled in an (automated) way from the original process model, which enables the distributed execution of each fragment by different process partners.

A problem that is left open is how process evolution and process model change is performed in these distributed process execution environments. A critical challenge for any process-aware information system (PAIS) is its ability to respond effectively to process changes (Weber, Sadiq, & Reichert, 2009; van der Aalst W. , 2001; Rinderle, Reichert, & Dadam, 2004). Processes evolve over time and the execution environment should adequately support these changes. The distributed execution environment adds, however, additional difficulties in process change support: a global process overview is unavailable since execution is fragmented, instances are created for fragments and not for the global process model and extra overhead is introduced since coordination between physically distributed fragments is needed to propagate changes in the execution environment.

In this paper we describe the difficulties, advantages and issues of process evolution in a fragmented and distributed process model execution environment and propose a system to propagate and coordinate process changes at runtime. The change support system is based on our previously proposed approach for distributed process execution, where a non-intrusive fragmentation algorithm is used; with dedicated, lightweight process engines for fragment execution, and an event-based communication paradigm between process fragments to ensure a scalable, loosely coupled and flexible runtime environment (Hens, Snoeck, Poels, & De Backer, 2012). Our previous work has demonstrated that beyond a certain system load threshold, the distributed process execution outperforms the centralized execution and the performance increase outweighs the distribution overhead. The fragmentation itself is kept non-intrusive, because this ensures that the process modeler does not have to know the technical details of process distribution and of the runtime architecture. In the same manner, the process redeployment (or change) system should also be non-intrusive, and allow for the automatic propagation of changes in the runtime architecture.

Throughout the rest of the paper, the term 'global process model' refers to the original (whole) process model; 'process model fragment' refers to an item of the outcome of the fragmentation algorithm and 'global process instance' and 'process fragment instance' refer to instantiations of the global process model and process model fragment respectively.

In this paper we follow a nested problem-solving approach as proposed by Wieringa (2009), which describes an intertwinement of practical and knowledge problems in the research cycle. At the top, we solve the practical problem of process evolution in the distributed environment. First, a running example is presented, which is used throughout the paper. Subsequently, our distributed event-based process execution approach is briefly described: its concepts, the fragmentation transformation and the execution architecture. In the second part of the paper, a knowledge problem is tackled first. The aspects of process evolution in such a distributed environment are discussed and issues are identified which complicate change management for fragmented and distributed processes. In the next section, a solution for the practical problem of change management in the distributed environment is presented: a protocol which describes every step involved in detail and addresses the issues identified. Finally, an evaluation of the distributed change management is given in the form of a prototype implementation which shows the feasibility of the approach and a discussion on any disadvantages compared to classical process evolution techniques. The paper ends with a discussion on related work and some concluding remarks.

Running Example

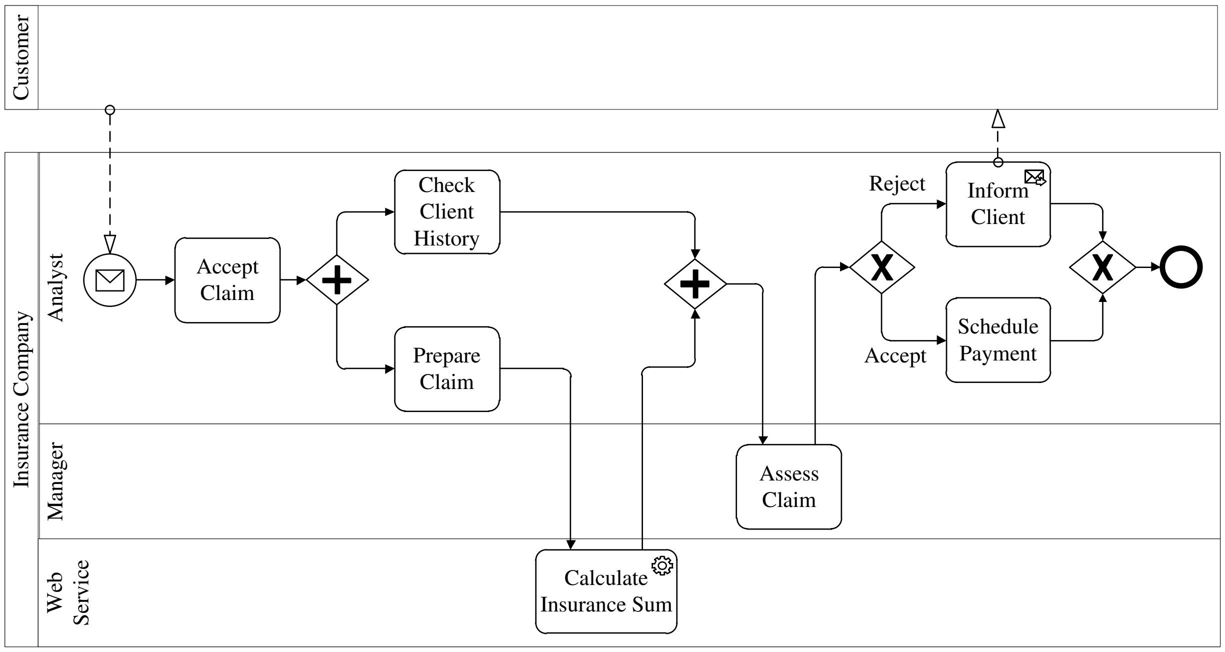


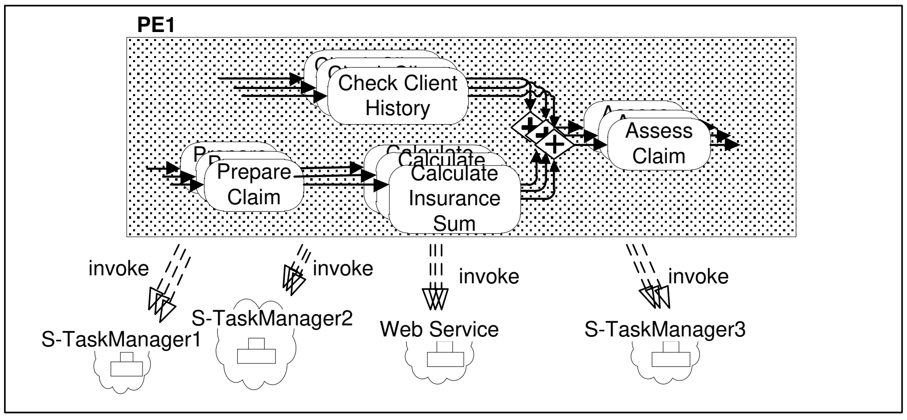
Figure 1. Running example.

Figure 1 shows a small BPMN business process diagram for insurance claim handling. Arriving claims are inspected by an analyst who prepares documents for the automated web service and creates a report of the client's history. These two tasks can be done in parallel by the analyst(s). A web service takes as input a prepared (computer readable) claim report and calculates an insurance sum based on all variables in the report. A manager in the insurance company takes the eventual decision based on the client's history report and the claim amount calculated by the web service. Finally, a sum is refunded or the claim is rejected. A process engine is used to execute and control the process flow, task managers are employed to handle the inbox for manual tasks and a web service is used to calculate the insurance sum.

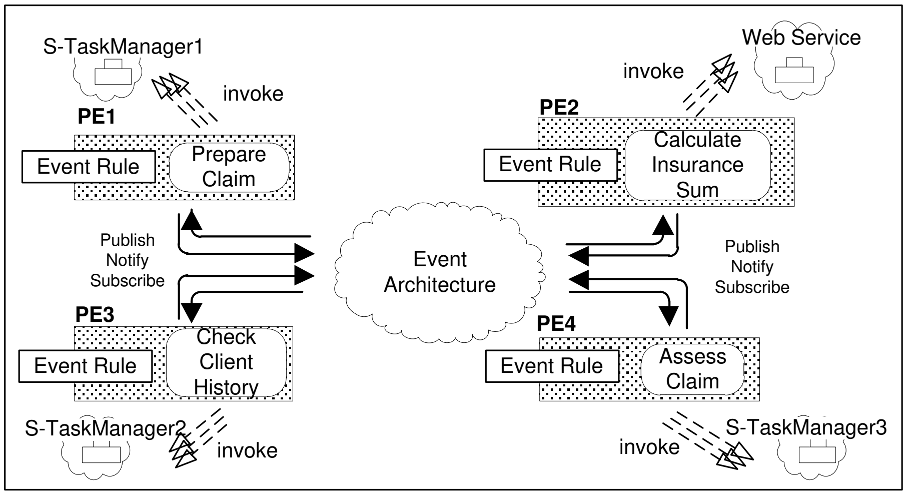
Although this example is kept small for exploratory purposes, the approach proposed in this paper is not limited to these simple examples. Also note that the example omits any data flow considerations. Since this research is focused on the control flow of process execution, for simplicity we assume data is transmitted along with the sequence flow.

Background: Distributed Event-Based Process Execution

In this section we give a brief overview of distributed event-based process execution and describe the fragmentation and execution architecture required to achieve the distributed execution.



(a) Centralized Process Execution



(b) Distributed Event-Based Process Execution

Process Engine
T Task
Service Component
Service Invocation
Process Flow

Figure 2. Distributed event-based process execution.

*Figure 2*a shows in part the centralized execution of the insurance claim handling process. One process engine (PE1) is used to manage the entire process flow, whereby multiple process instances are controlled at the same time. To overcome the managerial and technical issues of centralized execution, a distributed approach is adopted where not one, but multiple process engines are used to manage, in collaboration, the entire process flow. At deployment time a transformation algorithm takes as input the original process flow and outputs different process model fragments. These fragments are deployed on different, dedicated process engines as shown in *Figure 2*b. Instead of one (or multiple duplicated) engine(s), multiple engines are used which differ both physically (location in the IT architecture) and logically (execute a different process model fragment) from each other (see PE1-PE4 in *Figure 2*b). The global process execution for a certain process instance is performed by the collaboration of all the process engines, each running a different process model fragment.

To achieve this collaboration an event-based communication paradigm is used. In an event-based (publish/subscribe) communication architecture, components communicate with each other by generating and receiving event notifications (Mühl, Fiege, & Pietzuch, 2006). Components (*publishers*) publish notifications into the architecture, from where they are routed to other interested parties (*subscribers*). This routing is done by an event service that keeps track of which entity is interested in which event type and which entity is able to publish which event type (content-based many-to-many routing). In our case, the fragmented process engines are both publishers and subscribers of event notifications, with an event being a *past* happening in the PAIS with a business meaning, e.g. the completion of a business task, the arrival of a new task, the cancellation of a task, etc. Each notification hereby relates to the completion of a specific business task. For example: “Accept Claim completed for process instance 1” would be an event notification in the insurance handling’s process execution.

A consequence of using event-based communication is that for each process fragment it must be specified for which event types it needs a subscription (and thus a notification). This specification is called the *event rule* for the process fragment. The event rule specifies a logical combination of events that describes in which state the (business) environment should be before the process fragment can start executing. The process fragment is enabled if its event rule evaluates to true, i.e., the fragment’s process engine received event notifications indicating the completion of other fragments on which it is sequence dependent, or in other words: the environment is in a state where the fragment is able to execute. For example, the event rule for the ‘Assess Claim’ task from *Figure 1* will state that the task can start when ‘Check Client History’ `and` ‘Calculate Insurance Sum’ is completed (for the same process instance).

Figure 3 shows an overview of the major components of distributed event-based process execution. A process model is split into different process fragments, which are each executed by a different process engine. Each process engine is complemented with an event rule for which it needs subscriptions and one or more events which it publishes itself (i.e. completion events). To handle these publications and subscriptions, the engine is contained in a publish/subscribe wrapper which can communicate with the underlying event architecture. Eventually the process fragment's engine will create process fragment instances, publish event notifications and react to notifications, hereby collaborating with other process engines which results in the execution of the modeled, global process flow.

In the following sections the transformation from a global process model to process model fragments is briefly described, as well as the resulting distributed process architecture and execution.

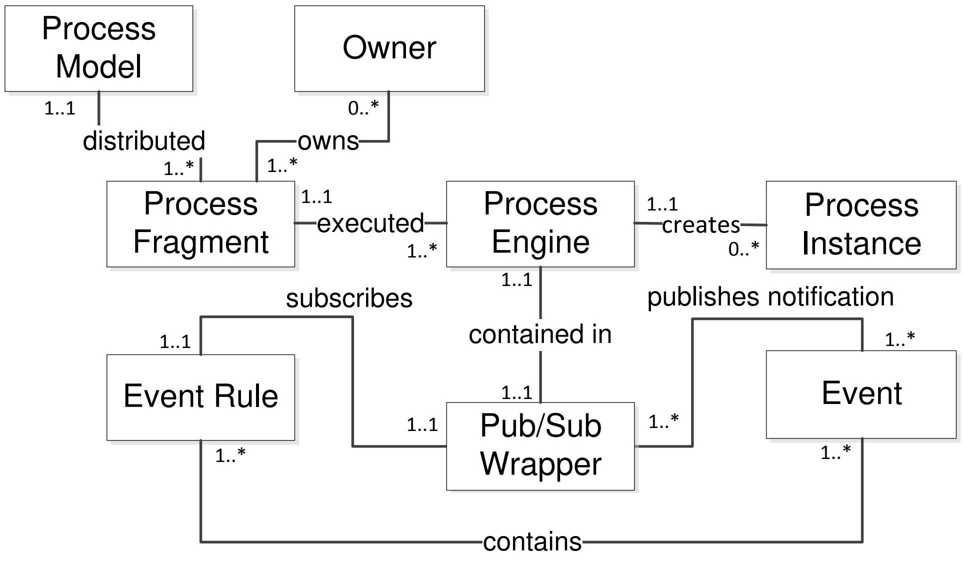


Figure 3. Meta model

Process Transformation

To be able to execute a process in the distributed event-based setting, the process model needs to be transformed into different process fragments, with each a corresponding event rule. Before the event rule for each fragment can be calculated, the fragments in the process model have to be defined by choosing a unit of decomposition. The unit of decomposition can be anything from a single task to a group of process model elements. Grouping of process elements is supported by *hierarchical process modeling*. Process model elements that form one fragment can be grouped in one *composite task* (or s*ub-process* in BPMN). The use of the concept of sub-process immediately provides the modeler with a correct fragmentation semantic and avoids the necessity of a separate fragmentation language. *Figure 4* shows an example of the insurance claim handling process fragmented into four fragments. At deployment, the transformation algorithm will use this grouping to split the global process model and calculate the event rules and completion events per fragment.

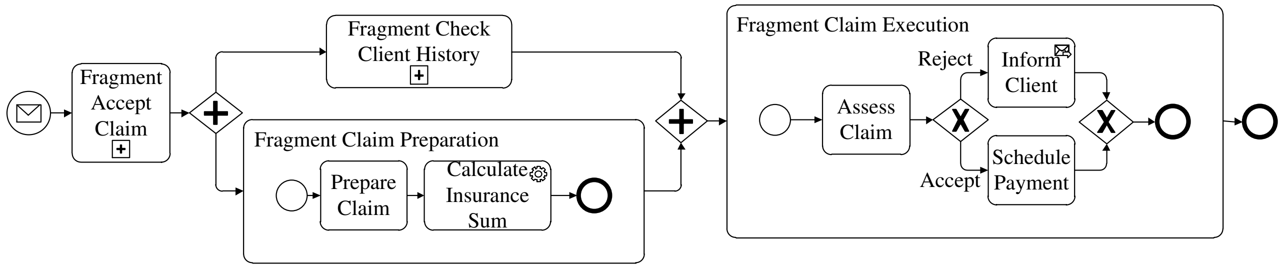


Figure 4. A fragmentation example

Grouping process model elements in composite tasks enables different fragmentation strategies. For example, a process model can be split according to workflow variants (Hallerbach, Bauer, & Reichert, 2010), according to the network traffic involved (Chafle, Chandra, Mann, & Nanda, 2004), according to user defined regions, etc. However, a restriction is imposed on the fragmentation by the event-based execution model (see *Figure 3*). The process model part contained in the fragment must be *structurally sound,* it has one entry point and one exit point. This is because each fragment can only contain one event rule and one completion event. A further discussion on this restriction is provided in the Related Work section.

Which fragmentation strategy is used is left out of scope of this paper. For illustration purposes, we assume that a task is used as unit of decomposition (similar to METEOR$_2$ (Miller, Palaniswami, Sheth, Kochut, & Singh, 1998)). Choosing a task as the unit of decomposition, guarantees a fine grained distribution of the global process flow. The presented transformation algorithm, enactment and process evolution protocol can be adapted accordingly to allow other decomposition units.

Transformation Algorithm. *Figure 5* shows the basic algorithm to split a global process flow (in $O(n^2)$ time). For illustration purposes, the algorithm only shows a very basic transformation, supporting a limited set of process modeling elements (tasks, sequence flows and exclusive- and parallel- gateways) and assumes a *task* as the unit of decomposition. An extended algorithm can be found in (Hens, Snoeck, Poels, & De Backer, 2012).

The algorithm takes as input a fully specified process flow and outputs different (executable) process fragments. Each resulting process fragment consists of a starting rule, a task (or sub-process) to execute and an (end) event to publish the completion of the fragment (see line 3). A starting rule for a split process consists of an event part (the event rule) and a user-defined conditions part. Finding the event rule for a fragment equals to finding which preceding fragments in the process flow need to be completed before the execution of the fragment can start. The algorithm finds these completion events by means of a depth-first search in the upward flow in the global process model. The event rule is transcribed as a logical expression in Disjunctive Normal Form, where an event in the expression is a signal (or event notification) which is matched with a publication event of another fragment (see line 9). The conjunctions and disjunctions in the event rule match with the gateways preceding the fragment (see lines 12-18). If, for example, the fragment is preceded by a parallel gateway, the event rule will consist of a conjunction containing the publication events of the fragments preceding the parallel gateway. Hence, the gateway logic preceding a fragment is included in the fragment itself. This ensures that only events are published which have a business meaning (relate to a specific task or sub-process in the original process model). Events indicate the completion of a task, the receipt of a message, etc. and not for example that a parallel split in a process path happened, or that a conditional merge happened.

The second part of the starting rule consists of user defined conditions originating from XOR-splits. A fragment is only enabled when its event rule evaluates to true AND the respective conditions evaluate to true. When searching for the completion events for the event rule, any condition encountered on an XOR-gateway is also stored in the starting rule (see lines 14-15).

As an example, after transformation, the task ‘Schedule Payment’ from the insurance claim handling process will be transformed to a fragment containing:

$$EventRule = AssessClaimCompleted \wedge (Accept = True)$$

$$t = Schedule\ Payment$$

$$EndSignal = SchedulePaymentCompleted$$

Algorithm 1 Basic Transformation

```
Definitions
  Process = $<T, G, SF>$
    with T the set of tasks, G the set of gateways and SF the set of sequence flows
    $SF \subseteq [(T \cup G) \times (T \cup G)]$
  Event = $<id, Signal>$, with $E$ the set of all events
  Fragmented-Process = $<EventRule, t \in T, endSignal \in E>$
  EventRule = $(\{event\} \vee \{event\} \vee ... \vee \{event\})$
            = combination of events in Disjunctive Normal Form

Procedure split(Process P)
 1: for all Task t ∈ P do
 2:   ER = eventRule(t)
 3:   FragProcess(t) = <ER,t,SignalOf(t)>
 4: end for

Procedure eventRule(Task t)
 5: $F = \{(x, t') | (x, t') \in SF \wedge t' = t\}$
 6: return $\bigvee_{f \in F} eventRule(f)$

Procedure eventRule(SF (a,b))
 7: if a = Task then
 8:     $id{+}{+}$
 9:     return $<id, SignalOf(a)>$
10: else if a = StartOfProcess then
11:     return $<id, a>$
12: else if a = XOR-Gateway then
13:     $F = \{(x, a) | (x, a) \in SF\}$
14:     return $( \bigvee_{f \in F} eventRule(f)) \wedge$
15:         $conditions((a, b)))$
16: else if a = AND-Gateway then
17:     $F = \{(x, a) | (x, a) \in SF\}$
18:     return $\bigwedge_{f \in F} eventRule(f)$
19: end if

Procedure conditions(SF (a,b))
20: if $ExpressionOn(a, b) = default$ then
21:     $F = \{(a, x) | (a, x) \in SF\} \setminus \{(a, b)\}$
22:     return $\neg( \bigvee_{f \in F} ExpressionOn(f))$
23: else
24:     return $ExpressionOn((a, b))$
25: end if
```

Figure 5. Basic Transformation Algorithm

Execution Architecture

After the transformation, each process fragment can be deployed to a dedicated process engine and these can be distributed into the enterprise architecture together with a publish/subscribe middleware which handles the publication, subscription and notification messages. *Figure 6* positions these components in a service oriented architecture (SOA), featuring an enterprise service bus (ESB) as connectivity layer (Chapell, 2004). The publish/subscribe middleware is a part of the infrastructure services which provides the routing and subscription facilities for the event messages. Each process fragment engine is a composition service and invokes one or more business services in the ESB. Additionally, management and monitoring services can be added which allow for the observation and administration of the distributed process execution.

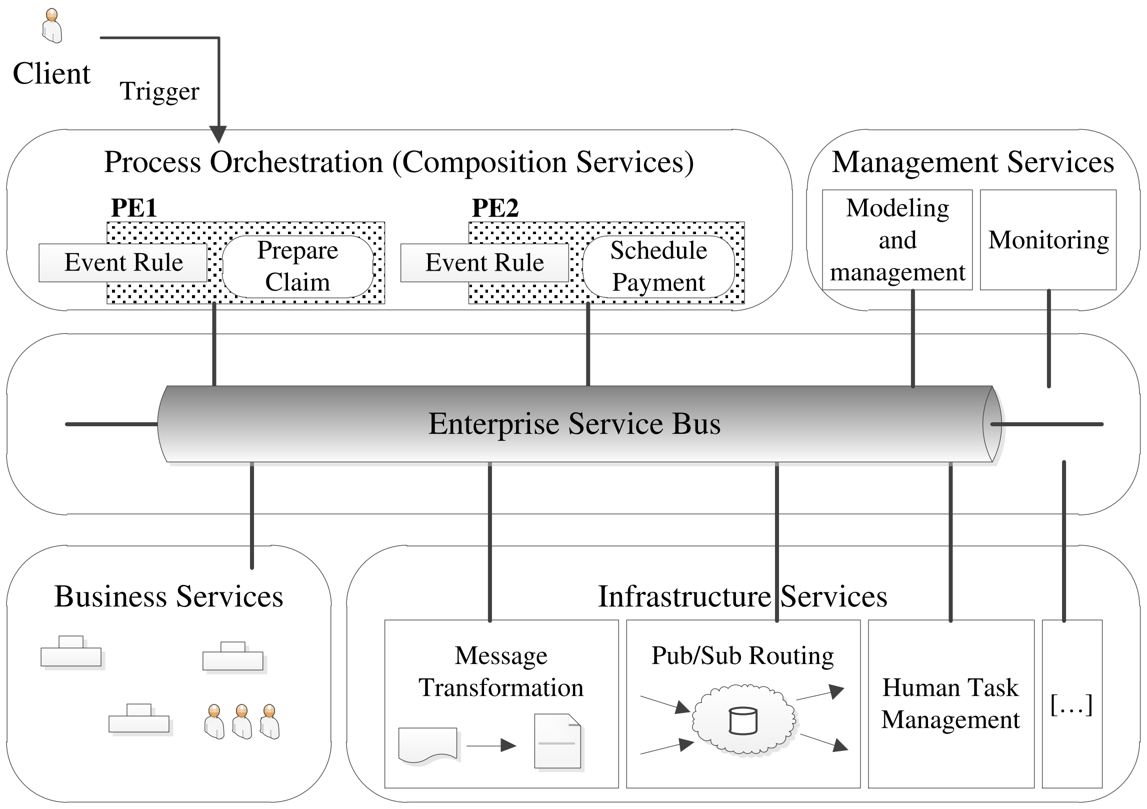


Figure 6. Distributed event-based process execution

Process Execution

Upon deployment into the enterprise architecture, each process fragment engine subscribes to the event types found in its corresponding event rule. For example, the fragment for task 'Check Client History' will subscribe to one completion event i.e. `AcceptClaim-`

`Completed.` After subscription, the fragment is capable of accepting event notification messages from the publish/subscribe middleware.

The data payload of an event notification message in the architecture should minimally consist of two things, one is the indication of the task it represents (i.e. the tasks name or id, which also serves as the event-id), and another is a process instance id, which indicates for which (global) process instance an action has been performed. The latter attribute is necessary not to lose the coupling between the global process instance and the action performed. Other attributes can also be added to allow for process specific data transmission or process control. An example event notification is:

```
[id="AcceptClaimCompleted";PIID="1";client-id="1234"]
```

Figure 7 shows the internal workings of a fragmented process engine embedded in a publish/subscribe wrapper and Figure 8 shows the possible states of a fragment instance. Upon arrival of an event notification three steps are performed by the fragmented process engine:

1. The notification is routed to the corresponding fragment instance. This is done by matching the instance id found in the event notification, with the instance ids of the already running fragment instances (in state evaluating). If no match is found, a new fragment instance is created. In the fragment instance, the event in the event rule corresponding to the notification will become enabled.
2. After each notification receipt, the event rule is evaluated. If the event rule evaluates to true, the state of the fragment instance changes from evaluating to ready (see Figure 8), indicating that the fragment is enabled and can start executing. When the instance is picked for execution (instantaneously for an automated business task), its state changes to ‘running’. An enabled fragment

instance can also move from the *ready* state back to the *evaluating* state, since for some workflow patterns (e.g. the milestone pattern (van der Aalst, Ter Hofstede, Kiepuszewski, & Barros, 2003)) it is possible that an enabled fragment becomes disabled again (Hens, Snoeck, Poels, & De Backer, 2012). If the fragment's execution is triggered, the original process engine takes over and executes the actual work described in the fragment (e.g. controlling a process model part, invoking a service task, etc.).

3. At the end of process execution (state *completed*), a notification is published by the publish/subscribe wrapper to signal the end of this fragment's instance, with the corresponding event-id and process-instance-id as attributes.

The published end event is routed through the event architecture (and the ESB), and picked up by other interested fragment engines, which handle this event again with the steps described above. Eventually, the combined execution of all these split process engines have achieved the global execution of the entire, designed process flow.

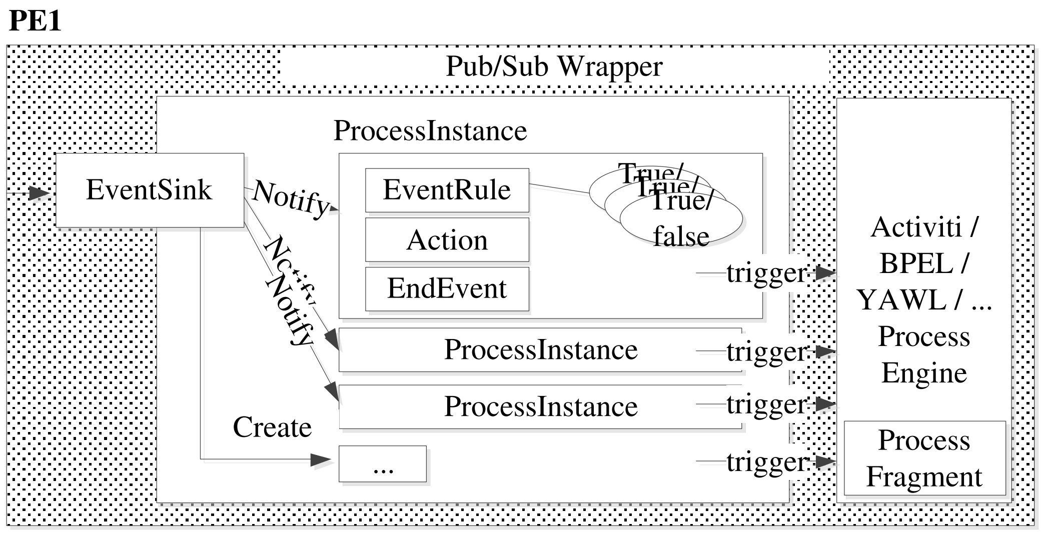


Figure 7. Fragment Process Engine

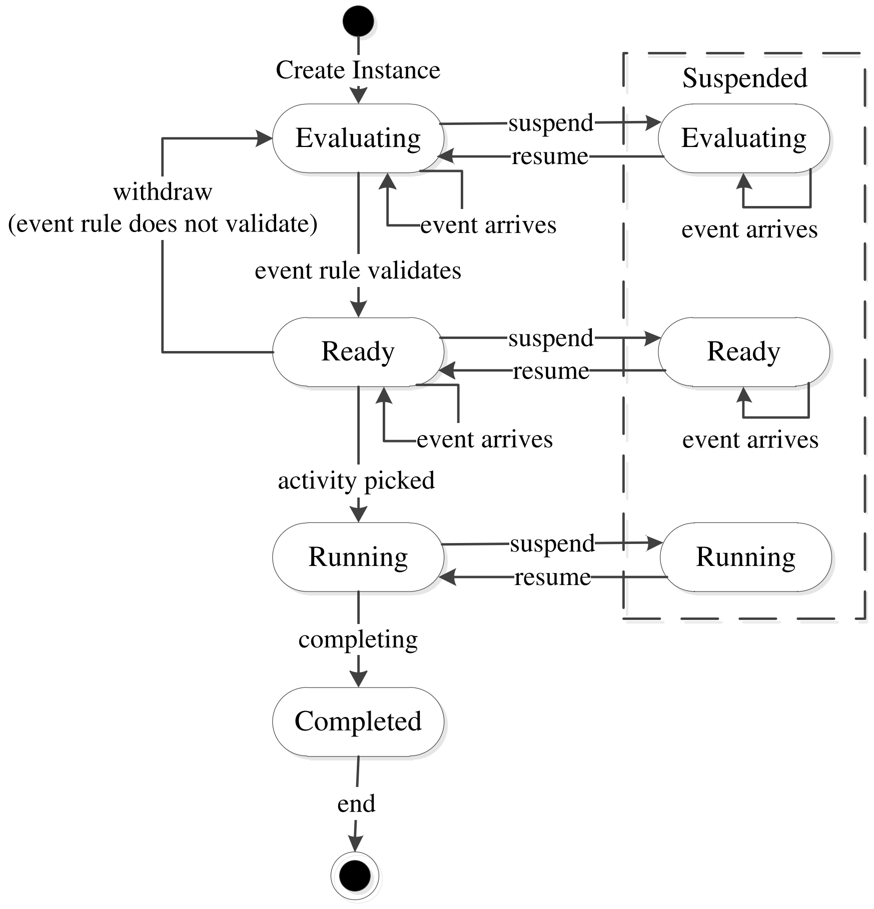


Figure 8. The state machine of a process fragment instance.

Process Evolution in the Distributed Process Execution Environment

The distributed execution environment described above allows the execution of a process model pre-defined at design time by different process partners and enables the distribution of process control, visibility and execution infrastructure. This (as-is) execution environment does, however, not allow any flexibility at process runtime. When a process is deployed for a certain process model, the execution will follow this pre-loaded model from start to finish. Such a non-modifiable execution is, however, often not sufficient for a PAIS (Weber, Sadiq, & Reichert, 2009): enterprises need to be able to quickly react to changing business needs and depend on the PAIS to support these changes.

Weber et al. (2009) give a good overview of flexibility issues in PAISs and present existing flexibility approaches along the phases of the process lifecycle. Flexibility support can be can be found in the modeling, execution and monitoring phases of the process lifecycle. Since this paper describes a process execution environment, we focus on flexibility in the execution phase, and more specifically on structural process model change support in the PAIS. We focus on the ability of the distributed execution environment to change the process's behavior by migrating a global process's specification to a new version. This is termed *evolutionary change.*

The next sections describe process evolution in more detail and define the difficulties involved when a distributed process execution scheme is used.

Process Evolution

Processes change over time, which means that one process type has multiple process schemas (Reichert, Rinderle, & Dadam, 2003). The relationship between process type, process schema and process instance is shown in *Figure 9*. Each schema represents another version of the process type. The PAIS should support the evolution of these process schemas by allowing the redeployment of a changed process model into the already running execution environment. After redeployment, any new process instance creation requests are handled with the new process schema. The biggest challenge for the PAIS is the management of the already running process instances during (and after) process change and redeployment. Especially for long running processes, it is possible that a number of process instances are being executed at the time of process change. Stopping these instances and rebooting them with the new process schema is not viable (due to task duplication). A versioning mechanism should therefore be adopted where at one moment, multiple instances belonging to multiple schemas for the same process type are running in the execution environment. The versioning mechanism keeps track of which process instances belong to which process schema and executes these instances accordingly (see *Figure 9*).

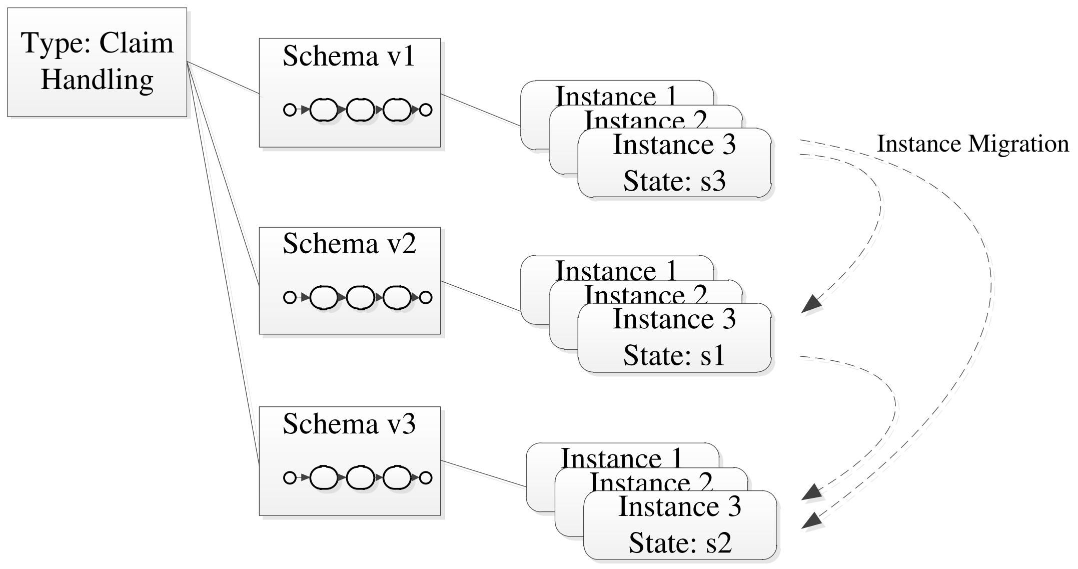


Figure 9. Versioning of a process type.

Adopting a simple versioning mechanism where only new instances are created with the new process schema is in some cases still not adequate. Most of the time the enterprise wants to migrate the maximum amount of process instances to the new schema version. For example, in the insurance claim process, a change is needed because a new regulation requires that a police report has to be filed before any claim analysis can happen. The addition of this extra condition has to be added to new (future) insurance claims as well as to existing, already started claims. The change has to be propagated to all running instances. This propagation is however not always possible. Current solutions define correctness criteria describing when a process instance can migrate to a new schema (Rinderle, Reichert, & Dadam, 2004). Globally, propagation of change can happen when the current execution state of the process instance is compliant with the new process schema. Several algorithms and techniques have been proposed which check and migrate process instances to a new process schema according to specific correctness criteria (Casati, Ceri, Pernici, & Pozzi, 1998; Reichert & Dadam, 1998; van der Aalst & Basten, 2002; van der Aalst W. , 2001). Most process evolution techniques can however not be used as-is in the distributed process execution environment. Additional difficulties are introduced by the fragmentation and distribution of the original process schema. These are discussed in the next section.

Difficulties in Distributed Process Evolution

Figure 10 shows the relation between the process schema, its fragments, the fragment's event rules and instances in a distributed process execution environment. Like in *Figure 9*, a process type has different schemas, each representing a different version of the global process. A process schema is transformed into different fragments at deployment time. Fragments are however not duplicated into the distributed environment according to the schema they correspond to. Fragments overlap across schemas. For example, in *Figure 10*, process schema v1 includes a task ‘Prepare Claim’, which is transformed into a single process fragment with a corresponding event rule (fragment 3). Process schema v2 also includes this task, but has a different event rule (different control flow dependency). Instead of duplicating the existing process fragment and assigning it another event rule, the same fragment is used and a new event rule is *added* to this fragment. Hence, a single process fragment can contain multiple event rules, each representing another schema version. In distributed execution, the multiple event rules of a process fragment represent the different versions of a process type in the actual runtime environment. An advantage of this approach where fragments are reused, but event rules are updated, is that only fragments for which the event rule changes need to be updated and redeployed. Fragments to which no changes apply can be left untouched and running in the architecture. In *Figure 10* for example, fragment 2 still uses its original event rule and serves all three process schemas.

The relation between process schema, fragment and event rule, which is inherent for the distributed execution, adds difficulties to process evolution: *late instantiation* of process fragments, *loss of the global process overview* and an *increased requirement for coordination* between process fragments and a change manager. These are discussed below.

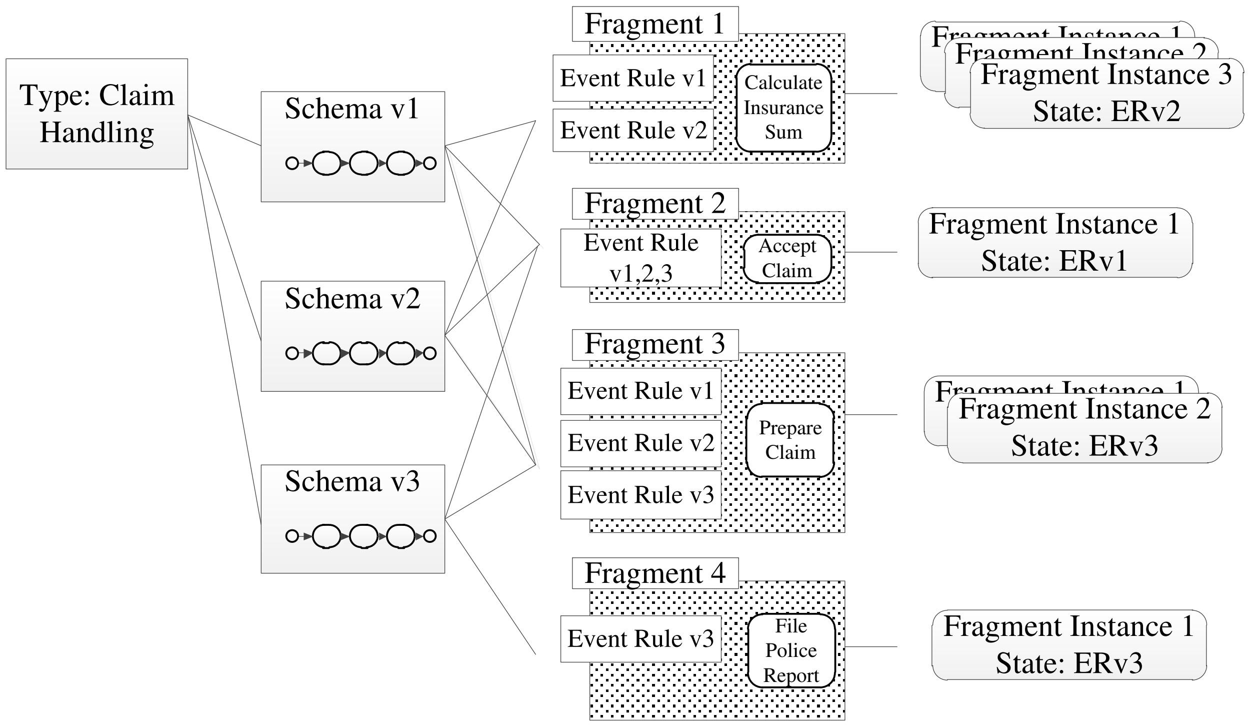


Figure 10. Versioning in the distributed execution environment.

Late instantiation. Fragments are instantiated 'just-in-time'. It is only upon activation, i.e. when an event notification is received, that a fragment is instantiated. Fragments that appear first in the process flow are instantiated earlier than fragments at the end of the process flow. *Figure 11* demonstrates the relation between the global process schema instance and its fragment instances. The global instance (top half) is in a state where task A is ‘running’. In the distributed environment (bottom half), there exists only one process fragment instance, which is an instance of fragment A. No instances have (yet) been created for the process model fragments in the downward flow of the process. Fragment B or C will only get instantiated after A completes and fragment D only instantiates after B or C completes.

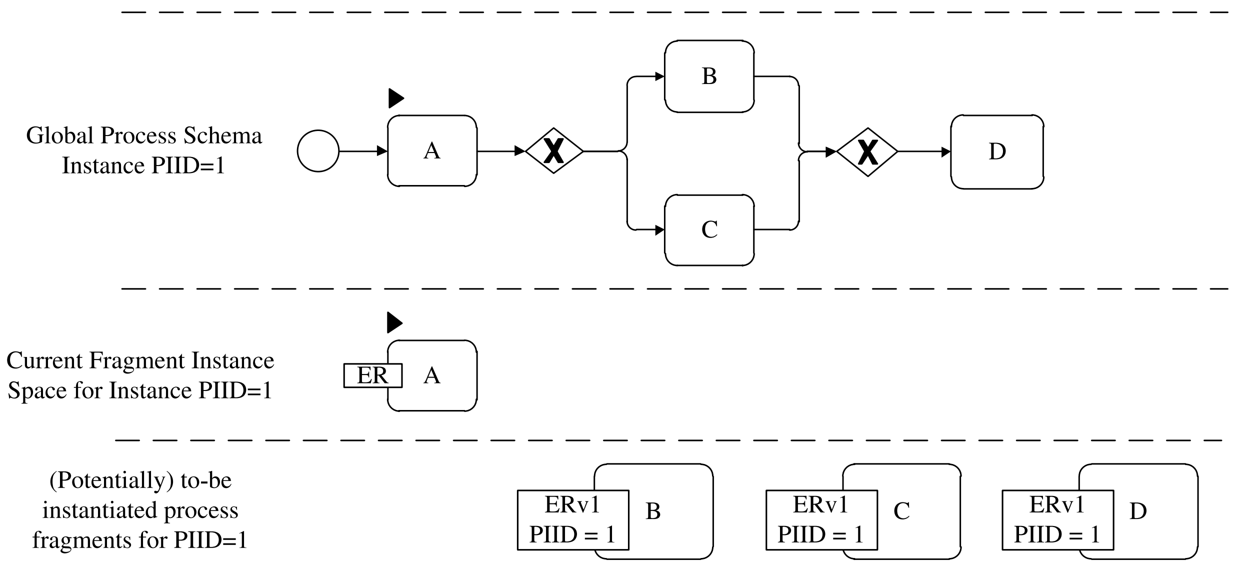


Figure 11. Late instantiation of process fragments.

Late instantiation has consequences for the versioning mechanism in the distributed environment. In a classical execution environment a process instance is simply linked to its corresponding schema version (see *Figure 9*). This is, however, not possible in the fragmented process execution. A versioning system cannot link a process instance up front to its corresponding schema version (i.e. event rule version), because not all process fragment instances exist yet (and no global process instance is available). A possible solution would be to instantiate every future process fragment (shaded fragments in *Figure 11*) ahead of time and link these to the correct event rule version, thus simulating the schema-instance link from *Figure 9*. The disadvantage is that fragments which will never be executed for a specific process instance also get instantiated, and stay in existence for the lifetime of the process type. This is the case when a conditional split is present in the global process flow. In *Figure 11*, both fragments B and C would get instantiated, whereas only one fragment instance will eventually get activated (B xor C). The other fragment instance stays in the *evaluating* state forever. This is problematic for big processes with multiple alternative paths, since these non-active fragments clog up resources over time. A distributed versioning mechanism therefore has to provide a way to cope with late instantiation.

Loss of a global process overview. In order to discover which running process instances can migrate to a new schema version, the state of the *global* running instance has to be inspected. For example, van der Aalst (2001) provides an algorithm which defines a region in a changed process model where instances which reside in a state within this region, are not able to migrate to the new version. Other change management techniques also need a notion of the state of the (global) process instance (Rinderle, Reichert, & Dadam, 2004). The problem in distributed execution is that the global overview of the entire process is lost. Each fragment executes its own process logic and has only knowledge of its own execution state. The global process state can only be reconstructed by inspecting the current state of each process fragment involved in a specific process execution. This requires a lot of communication between a central manager and the process fragments (certainly for large processes with a high amount of fragments).

Another way of reconstructing the state of the global process is enabling runtime logging of each event in the distributed execution environment. The resulting event traces can be used to calculate the current state of a specific process instance, given the corresponding process schema. The drawback of this approach is that a runtime monitor has to be available at all times, since missed events make it impossible to reconstruct the state of the process.

Need for additional coordination. Since the fragments are distributed, extra coordination is required to migrate changes to specific process fragments. A (reliable) communication protocol has to be established so that process fragments can react to control messages from a central change manager, which enables state-inspection, suspension, change propagation, etc.

A Distributed Process Evolution Protocol

In the previous section we introduced the difficulties of process evolution in a distributed process execution environment. In what follows, we propose a process evolution system which tackles those problems and shows the feasibility of change evolution in a distributed process execution environment. Since a method for instance migration is needed, the change region method from van der Aalst (2001) is described first. The second part of this section describes all steps involved in propagating a change into a running, distributed event-based process execution environment.

Change Region

Since changes in the process schema need to be migrated to the running process instances, a mechanism is needed which decides when a process instance can migrate and when it cannot. For illustration purposes the algorithm presented by van der Aalst (2001) is adopted in the distributed change protocol. Any other technique for change migration can also be used, provided the protocol presented below is adapted accordingly (see Related Work section).

The algorithm calculates a region in the old (and new) process schema, called *the change region*. The region contains a collection of states which cannot migrate to a state in the new process version. The change region is calculated with the following correctness criteria:

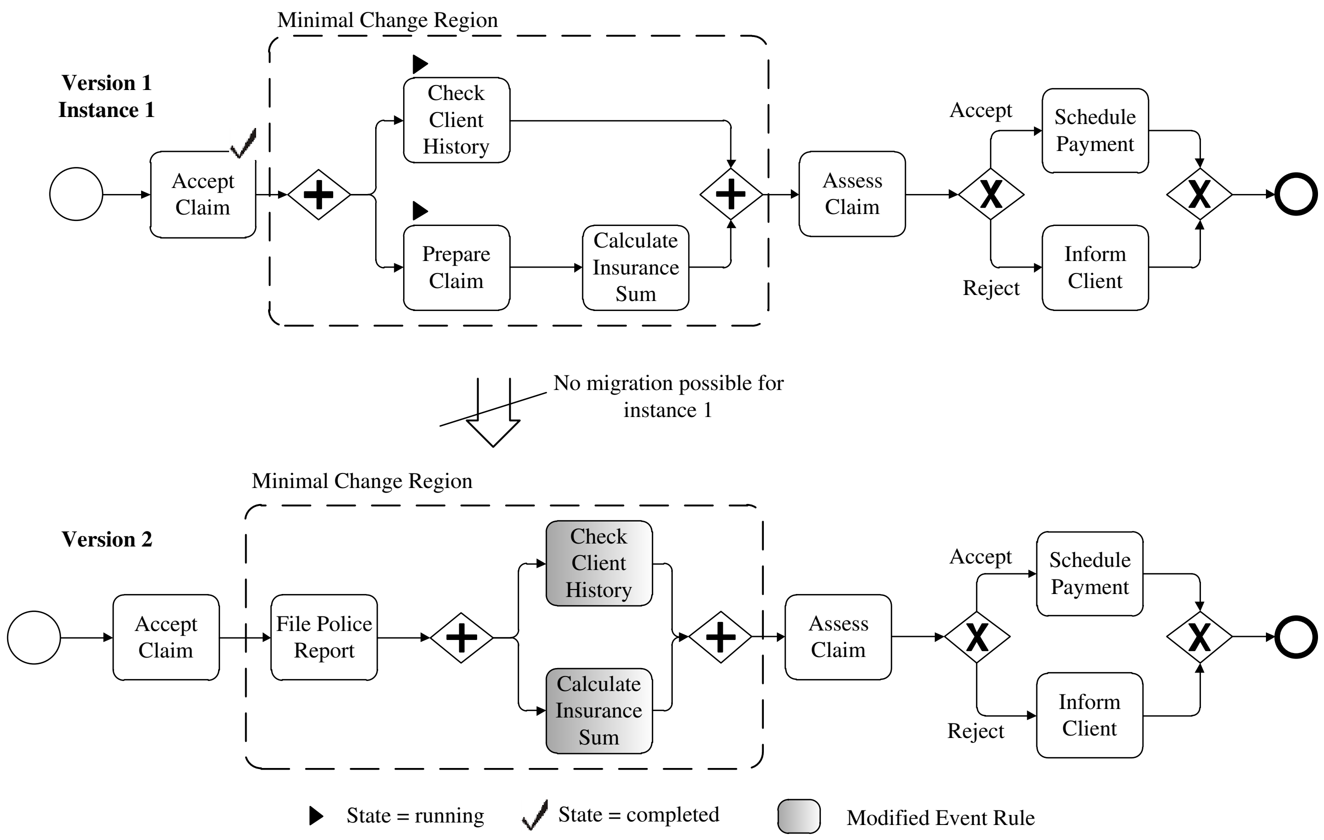


Figure 12. Change region for two versions of the running example's process.

"a transfer is valid if the state of the case after migration could have been reached from the initial state" (van der Aalst, 2001, p. 304). *Figure 12* shows two versions of the insurance claim handling process. Two change operations are performed: a task 'File Police Report' is inserted in the process flow; and the task 'Prepare Claim' is deleted. The change region in both versions is indicated with the dashed box. In general, the change region contains the modified flow elements in the process, supplemented with elements of any alternative paths originating from gateways, such as to create a block structured region. Any process instance that resides in an executable state in the region cannot be migrated to the new version and should be left running in the old configuration. For example, instance 1 shown in *Figure 12,* has an executable state inside the change region and can therefore not migrate to a new schema version.

Process Evolution Protocol

In this section we present the steps and communication messages involved for process evolution in the distributed environment. The redeployment protocol starts from the moment a process modeler defined a new process schema. When the modeler redeploys the schema, the deployment engine (change manager) takes the old and new process specification as input. From here on the old schema is referred to as $P^o =< T^o, G^o, SF^o >$ and the new schema is referred to as $P^n =< T^n, G^n, SF^n >$ (see *Figure 5*). Every step is described in detail below[1]. The difficulties in process evolution: loss of the global process overview and late instantiation are tackled in steps 3 and 5 respectively and the additional coordination messages are described in the proof-of-concept section. Note that the protocol is used to realize changes in the inter-fragment structure. If a change resides inside a fragment itself, classical centralized process evolution techniques can be used to change and migrate the process fragment instances without interference with other fragments.

Step 1: Change region. The dynamic change region is calculated for P^o and P^n. The outcome is a set of process flow elements: CR, which make up the change region.

Actor: Change Manager

$$CR = calculateChangeRegion(P^o, P^n)$$

$$CR \subseteq (T^o \cup T^n)$$

Figure 12 shows the change region for the two versions of the insurance claim handling process: $CR = \{ 'check\ client\ history',\ 'prepare\ claim',$

$$'calculate\ insurance\ sum',\ 'file\ police\ report'\}$$

Step 2: Suspension. The execution of every fragment, which has instances residing inside the change region is suspended. A control message is sent to each fragment in the change region, which suspends every *evaluating*, *ready* or *running* process fragment instance (see *Figure 8*). Additionally the fragments just outside the border in the upward flow of the change region are also suspended. The latter is necessary to prevent process instances which are in an executable state outside the change region entering an executable state inside the change region during process redeployment. If such a state change would happen between steps 3 and 4 of the redeployment steps, the process instance will reside in an inconsistent state.

Actor: Change manager and process fragment engines

$$ToSuspend = \{x \mid (x \in CR \cap T^o) \vee (x \in BorderFragment)\}$$

$$BorderFragment = \{x \mid x \in T^o \wedge (x, y) \in SF^o \wedge y \in CR\}$$

Suspending the fragments inside the change region and the border fragments in the upward flow is the minimal suspension criteria for process redeployment. Every other fragment can keep executing fragment instances. This has as advantage that process instances which are in an execution state past the change region can execute and terminate without downtime. Non-border fragments before the change region can also keep executing, but any events received by a suspended fragment will be queued and only propagated to the fragment instance after redeployment is complete. In contrary to centralized process execution, downtime during change propagation is thus minimized to instances inside the change region.

From the global process schemas P^o and P^n of the insurance claim handling process (*Figure 12*), the change manager deduces that the fragment engines running the tasks 'Check

Client History', 'Calculate Insurance Sum', 'Prepare Claim' and the border fragment 'Accept Claim' should suspend their actions. A control message is sent to these fragment's engines, where the engine will suspend each fragment instance[2]. Instances already in an execution state past the change region ('Assess Claim' and downward) are not suspended. *Figure 13* shows the fragment's engines of the insurance process. PE1, PE2, PE3 and PE4 are suspended. Note that the situation is only shown for process instance 1 of *Figure 12;* many other process instances can be running at the same time.

Step 3: Execution state retrieval. Since process instances which reside in an executable state inside the change region cannot migrate to the new schema version, the next step is to determine these process instances. As mentioned in the previous section, the state of the global process is not readily available in the distributed environment. The advantage of the use of the change region is that not all fragments in the execution environment need to be inspected, but only those belonging to the change region. A global process instance has an executable state inside the change region if it is executing a task belonging to this region. Any fragment instance in a *running* or *ready* state inside the change region thus indicates that its respective global instance also resides in the change region.

A control message is sent to each fragment in the change region, requesting the process instance ids (PIIDs) of instances which are currently in the (suspended-) ready or (suspended-) running state.

Actor: Change manager and process fragment engines

$$PIIDs = \{y \mid y \in getRunningProcessInstances\ (x) \land x \in CR\}$$

$$getRunningProcessInstances : T \rightarrow \{\mathbb{N}\}$$

$$x \rightarrow \{PIID \mid y \in instancesOfFragment(x)$$

$$\land\ (state(y) = ready \lor running \lor evaluating)$$

$$\land PIID = idOf(y)\}$$

A second option to retrieve the process instances which are not allowed to migrate and which does not require the inspection of individual fragments, is the inspection of the event traces captured by a monitor (if available). For example, the event trace of instance 1 in *Figure 12* is: `AcceptClaimCompleted`. This trace can be mapped on the global process schema from which it can be concluded that the global process instance is in a state where 'Check Client History' and 'Prepare Claim' are enabled (or executing) and thus falls inside the change region. Each process instance trace can be checked and PIIDs can be collected.

For the insurance claim handling process, the process fragment engines PE2 ('Prepare Claim'), PE3 ('Check Client History') and PE4 ('Calculate Insurance Sum') are requested to respond with the PIIDs of their ready and running fragment instances. PE2 will respond with the set $\mathrm{PIIDs} = \{1\}$, PE3 responds with the same set and PE4 responds with an empty set since no instance fragments are present at the time of inspection (see the late instantiation section). The combination of these responses are the PIIDs of global instances residing inside the change region, i.e., $\mathrm{PIIDs} = \{1\}$.

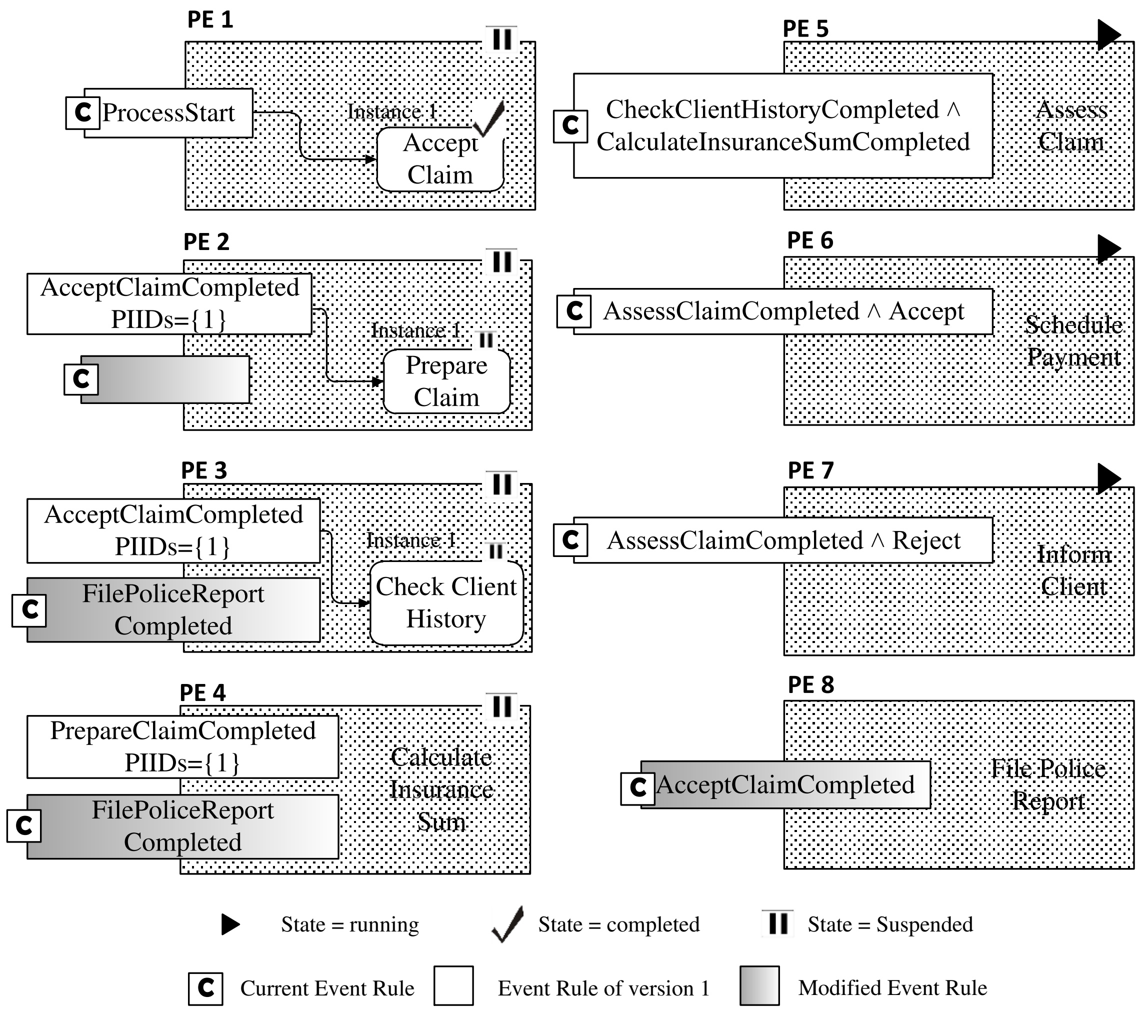


Figure 13. The distributed execution environment for the insurance claim handling process in a state between steps 5 and 6 of the redeployment process.

Step 4: Process fragmentation. Each process fragment where the event rule changed needs to be updated. The new process schema (P^n) is therefore transformed with algorithm 1 (see *Figure 5*) into different fragments with their own event rule. Since only fragments that changed need to be updated, the fragments of the new schema are compared with the fragments of the old schema version. The difference between the old and new fragments are the fragments which need to be redeployed. Additionally, tasks which are removed from the new schema version also need to update their event rule. The deleted fragment has to unsubscribe to any events it previously subscribed to (unless it still participates in 'old' process instances, see step 5). Note that the set of to-be-redeployed process fragments is by definition a subset of the change region (van der Aalst, 2001), but is not necessarily equal to the change region.

Actor: Change manager

$$\Delta = \left(split(P^n) \setminus split(P^o)\right) \cup \{\forall\, x \in (T^o \setminus T^n) :\ emptyRule(x)\}$$

$$emptyRule: T \rightarrow < ER, T, Event >$$

$$x \rightarrow < , x, >$$

In *Figure 12*, 'Check Client History' and 'Calculate Insurance Sum' have a changed event rule and need to be redeployed. Additionally, since 'Prepare Claim' is removed from the new process schema it is also added to the to-be-redeployed process fragments. The inserted task 'File Police Report' is deployed as usual.

Step 5: Fragment redeployment and versioning. Each process fragment engine that has to change its event rule is sent a redeployment control message including the new event rule and the set of PIIDs that still need to run on the old schema version (i.e. event rule).

Actor: Change manager

$$\forall\, x \in \Delta :\ redeploy(x, PIIDs)$$

One of the specific problems of distributed processes is the late instantiation of process fragments. As a result of this problem, not every fragment instance can immediately be linked to its corresponding event rule version. This is the case for PE4 where no fragment instance is instantiated yet for process instance 1 (see *Figure 13*). However, PE4 has to eventually execute instance 1 with the old event rule version (it falls within the change region). To circumvent this difficulty, a list of event rules is kept in each process engine, which links a rule to a set of PIIDs and links one rule to a 'current' tag. Upon receipt of the redeployment message, the fragment's engine links its current rule to the PIIDs included in the message and makes the new event rule the current one.

Actor: Process fragment engines

$$EventRules = \{(rule, \{PIID\} \vee \texttt{current}) >\}$$

Figure 13 shows the situation of the fragment engines after redeployment. Engines PE2, PE3 and PE4 contain changed event rules (the shaded rectangles). The old event rule is linked to a set of process instance ids (in this case only one id) and the new event rule is the current one.

Because now a fragment has multiple event rules (versions), step 1 from the execution semantics of the fragment engine changes slightly. Upon arrival of an event notification message, it is checked if this event should be handled with the current rule or an old rule. An old event rule is used if the instance id included in the event notification message is also included in a PIID set of the `EventRules` list. If the id is included in the list, the corresponding event rule version is used to instantiate a fragment instance. For PE4 in *Figure 13*, upon receipt of a `PrepareClaimCompleted` event notification for $instance = 1$, the engine will create a fragment instance linked to the old event rule. Any notification corresponding to other process instances will be linked to the current, new event rule.

Step 6: Resumption. The last step in the redeployment process is resuming all suspended process fragment engines.

Actor: Change manager

$$\forall\, x \in ToSuspend : resume$$

The protocol described above allows the migration of instances compliant with the new process schema. In some situations, non-compliant instances should however also be allowed to migrate. Rinderle-Ma et al. (2010) discuss different migration strategies for non-compliant instances. Their second strategy: adjusted instance specific changes, can be used in the distributed environment to allow for non-compliant migration. The only extra requirement is that a system for ad-hoc changes is made available in the distributed environment (see Future Research).

Proof of Concept Implementation

The proposed change protocol can be realized in the ESB architecture, by enabling a central change manager which performs the steps described. The change manager only needs as input the original and new process model. *Figure 14* shows the communication protocol between the change manager and a process fragment. This coordination requires the use of *control event messages*. These are event messages still following the publish/subscribe paradigm, but having a predefined format describing a certain action that has to be performed. A process engine subscribes to the control event messages next to its regular process related events. Upon receipt of a control message an interpreter reads the actions included and performs the necessary steps. The control event message should minimally consist of an indication that it is a control message, the fragment for which the message is intended (location-independent fragment-id) and the actual action that needs to be performed with any necessary attributes. Similarly, fragment engines can send control messages back to the change manager.

For example:

```
[id="CTRL";fragment="PrepareClaim";action="suspend"]
[id="CTRL";fragment="CheckClientHistory";
 action="redeploy";event-rule="FilePoliceReportCompleted";
 PIIDS="1"]
[id="CTRL";fragment="ChangeManager";action="PIIDS";
     PIIDS=”1,2,3,4”]
```

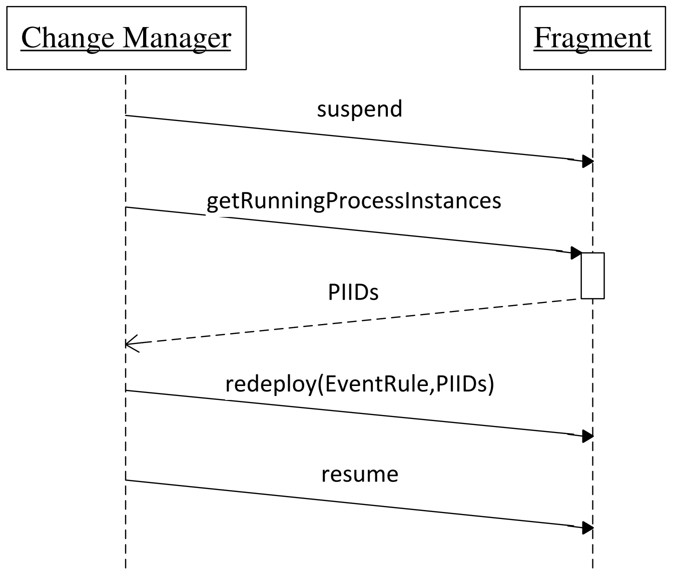


Figure 14. Interaction between change manager and process fragment.

The advantage of reusing the publish/subscribe architecture is that no extra service has to be provided which coordinates the change propagation actions. Every communication is handled by the publish/subscribe architecture which knows the location of each fragment engine and can therefore easily forward messages. If a more standardized and WS-* based approach is needed, the change propagation protocol can be plugged into a standard coordination framework like WS-Coordination (OASIS, 2009).

To demonstrate the feasibility of the approach, we developed a prototype of the distributed execution environment (Hens, Snoeck, Poels, & De Backer, 2012) and implemented a change manager based on the process evolution protocol described above. The prototype is based on BPMN2.0 process schemas with the Siena publish/subscribe service (Carzaniga, Rosenblum, & Wolf, 2001), the Activiti BPMN2.0 process engine (Alfresco, 2012) and event control messages as previously described. A change manager is made available which allows process evolution and controls the migration of process instances to a new process version.

The major disadvantage of the distributed change propagation compared to change management techniques in a classical process execution scenario (see *Figure 2*a) is the added network overhead when redeploying a changed process schema. For a specific process change the overhead can be calculated as follows:

$$network\ overhead = 4 * |DC| + 2 * |BorderFragment| + |\Delta|$$

Four messages are sent to and from each fragment in the change region (`suspend`, `getRunningProcessInstances`, `PIIDs` and `resume`), one extra suspend and resume message is sent to the border fragments and the redeploy message is sent to each fragment of which the event rule changed. For a simple change pattern like a 'serial insert' (Weber, Reichert, & Rinderle, 2008), the redeployment coordination already takes 7 messages (see *Figure 15*). Coordination messages remain very small (~100bytes), but still have to traverse the network, which can delay the redeployment process.

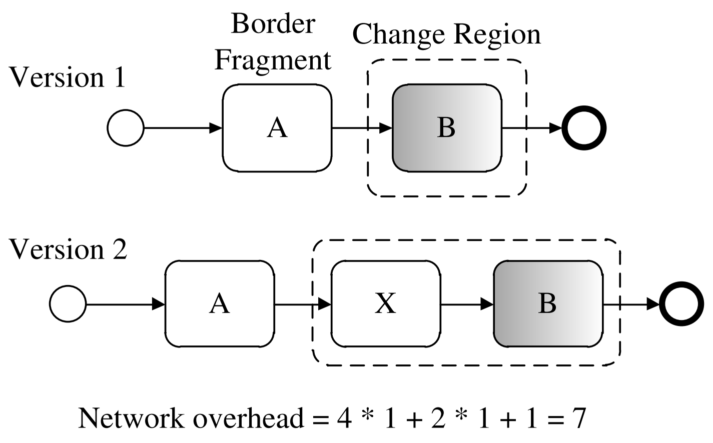


Figure 15. Network overhead of the serial insert change pattern.

The correctness of the evolution protocol is validated by checking the correct execution of a process instance during process evolution for all supported change patterns (Weber, Reichert, & Rinderle, 2008). To this end, our prototype implementation is used to simulate the described change and to check the correct execution of the instance after change propagation. The protocol has also been experimentally evaluated for its performance, robustness and scalability. A detailed explanation of this evaluation can be found in (Hens, Snoeck, & De Backer, 2012).

Related Work

Distributed Workflows. In the domain of PAIS, the problem of centralized process execution is recognized by many researchers (Muth, Wodtke, Weissenfels, Dittrich, & Weikum, 1998; Fdhila, Yildiz, & Godart, 2009; Chafle, Chandra, Mann, & Nanda, 2004; Li, Muthusamy, & Jacobsen, 2010; Fjellheim, Milliner, Dumas, & Vayssiere, 2007; Khalaf, Kopp, & Leymann, 2008). All approaches identify situations where a centralized process execution environment is not adequate. As described in the introduction, fragmentation is done with different technologies and according to specific criteria: to reduce network overhead, to increase availability and failure resilience, or to enable the distribution of fragment coordination and ownership. The use of event architectures in distributed process execution is also proposed by Li et al. (2010) and Fjellheim et al. (2007). Li et al. (2010) use the publish/subscribe architecture to increase flexibility, scalability and adaptability of the process flow and Fjellheim et al. (2007) use the publish/subscribe architecture to target mobile devices.

A restriction of our approach is that each process model part contained in a fragment should be structurally sound (comparable to the MENTOR approach (Wodtke, Weissenfels, Weikum, & Dittrich, 1996)). This is in contrast with e.g. ADEPT$_{distribution}$ (Bauer & Dadam, 1997) which does not restrict the fragment's definition. A way to permit arbitrary distribution in our approach is to change the meta-model (see *Figure 3*) and allow multiple event rules per fragment, one per fragment entry point; and multiple end events, one per fragment exit point.

Adaptive Process Management. In classical centralized process execution environments, change management is thoroughly researched: Weber et al. (2009) provide a good overview of this domain. Specifically for flexibility in the execution phase of the process lifecycle, a lot of different techniques have been proposed to migrate running process instances to a new schema version (Casati, Ceri, Pernici, & Pozzi, 1998; Reichert & Dadam, 1998; van der Aalst & Basten, 2002). These techniques differ by the correctness criteria used to propagate changes in the runtime process execution environment. Rinderle at al. (2004) provide a good overview of the commonly used correctness criteria and respective change management techniques. The change technique described here uses a specific correctness criteria with a corresponding change algorithm (van der Aalst W. , 2001), but can be extended to be used with other criteria as well (based on e.g. change operations instead of a change region).

There are a few approaches that discuss the combination of process changes and distributed workflow execution. Kochut et al. (2003) present the IntelliGen approach. In a similar way as our technique, they allow the individual control of process fragments, enabling updates of the fragment's specification. The approach does however not discuss how to migrate running process instances and only allows new cases to be run in the new process configuration. Casati et al. (1998), Weske (1998) and Barbará et al. (1996) also describe a distributed execution environment and additionally describe workflow evolution, but do not explicitly address how the two interact. Instance specific changes in a distributed environment are enabled by $\text{ADEPT}_{\text{distribution}}$ (Reichert & Bauer, 2007). Similar to our approach, only relevant workflow servers are inspected and synchronized, keeping communication costs to a minimum. However, different from our approach, $\text{ADEPT}_{\text{distribution}}$ assumes a local copy of the complete process schema at each distributed node. Rinderle et al. (2006) present an evolution technique for process choreographies. Process choreography evolution can be compared to our approach as a choreography also describes a process between distributed entities (organizations). The difference lies in the fact that our approach focusses on internal business processes which need to be distributed, instead of a change in the message interaction between public processes and how this affects internal, private processes.

Event Rules. The combination of event rules (in e.g. Event-Condition-Action format) and processes is also described in Event Driven Service Oriented Architectures (Levina, 2009). Their focus is however on the invocation of services, not on the decentralization of the process flow. In declarative process modeling, event rules are also used to represent a process flow. In DECLARE (van der Aalst, Pesic, & Schonenberg, 2009) rules identify the enablement of a process part. Kappel et al. (1997) also effectively implement Event-Condition-Action rules for workflow management. The biggest difference is that we start from an imperative process model and transform this to a distributed model with declarative rules, while in declarative modeling, models are defined in a declarative way from the start.

Conclusion and Future Research

To increase availability and performance and enable the distribution of control and/or visibility of a process specification, a distributed process execution architecture can be adopted. Many process fragmentation solutions are proposed which fragment and distribute a given process specification among different process partners. These solutions do, however, leave the dynamic features of a process enactment system out of consideration. This paper outlined the difficulties in process evolution in a distributed environment and demonstrates that a viable solution exists for these problems. Late instantiation and the loss of the global process overview in a distributed setting demand for extra coordination. We developed a protocol which is able to cater for the specific problems of process evolution in a distributed environment and showed the different steps needed to support the propagation of changes in the schema level to changes in the instance and fragment level.

One major advantage of change management in the distributed environment is that process instances are not suspended as a whole during change propagation: only relevant fragments are suspended. On the other hand, a drawback of the approach is that it generates extra network overhead. Nevertheless, the decision on adopting a distributed process environment should not be based only on the results of an extra communication cost during process change management. This paper explains that process evolution can be supported in the distributed environment. Advantages of the distributed runtime environment outweigh any disadvantages in the redeployment process of that execution environment (i.e. performance increase during process runtime versus network overhead during process redeployment). When an organization requires highly available, quick response process systems, a distributed process execution approach can be adopted, which can leverage the processing power from the distributed environment.

Future research involves implementing other dynamic features (Weber, Sadiq, & Reichert, 2009), which provide an added value for distributed process execution. The protocol described here supports top-down changes, where the global process flow is changed and redeployed. Since process fragments can each be managed by other process partners, it is also interesting to look at bottom-up changes. A fragment's manager can decide to change the starting (event) rule of his fragment, hereby implicitly also changing the global process flow. In addition, ad-hoc or instance specific changes in the distributed process environment can also be investigated.

References

Alfresco. (2012). *Activiti BPM Platform*. Retrieved December 5, 2011, from http://www.activiti.org/

Barbara, D., Mehrotra, S., & Rusinkiewicz, M. (1996). INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management, 7*(1), 5-15.

Bauer, T., & Dadam, P. (1997). A distributed execution environment for large-scale workflow management systems with subnets and server migration. *International Conference on Cooperative Information Systems* (pp. 99-108). Kiawah Island: IEEE.

Carzaniga, A., Rosenblum, D. S., & Wolf, A. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems, 19*(3), 332-383.

Casati, F., Ceri, S., Pernici, B., & Pozzi, G. (1998). Workflow evolution. *Data & Knowledge Engineering, 24*(3), 211–238.

Chafle, G. B., Chandra, S., Mann, V., & Nanda, M. G. (2004). Decentralized orchestration of composite web services. In S. Feldman (Ed.), *13th international World Wide Web conference on Alternate track papers & posters* (pp. 134-143). New York: ACM.

Chapell, D. (2004). *Enterprise Service Bus: Theory in Practice.* Sebastopol: O'Reilly Media.

Fdhila, W., Yildiz, U., & Godart, C. (2009). A Flexible Approach for Automatic Process Decentralization Using Dependency Tables. In P. Hofmann (Ed.), *IEEE International Conference on Web Services* (pp. 847-855). Los Angeles: IEEE Computer Society.

Fjellheim, T., Milliner, S., Dumas, M., & Vayssiere, J. (2007). A process-based methodology for designing event-based mobile composite applications. *Data & Knowledge Engineering, 61*(1), 6-22.

Hallerbach, A., Bauer, T., & Reichert, M. (2010). Configuration and management of process variants. *Handbook on Business Process Management, 1*(1), 237-255.

Hens, P., Snoeck, M., & De Backer, M. (2012). Measuring the Impact of Suspension on the Process Enactment Environment during Process Evolution. In G. Poels (Ed.), *International Conference on Research and Practical Issues of Enterprise Information Systems* (p. Accepted). Gent: Springer.

Hens, P., Snoeck, M., Poels, G., & De Backer, M. (2012). Process Fragmentation, Distribution and Execution using an Event-Based Interaction Scheme. *Submitted to the Journal of Systems and Software*.

Kappel, G., Rausch-Schott, S., & Retschitzegger, W. (1998). Coordination in workflow management systems - a rule-based approach. *Coordination Technology for Collaborative Applications, 1364*, 99-119.

Khalaf, R., Kopp, O., & Leymann, F. (2008). Maintaining Data Dependencies across BPEL Process Fragments. *International Journal of Cooperative Information Systems, 17*(3), 259-282.

Kochut, K., Arnold, J., Sheth, A., Miller, J., Kraemer, E., Arpinar, B., & Cardoso, J. (2003). IntelliGEN: A distributed workflow system for discovering protein-protein interactions. *Distributed and Parallel Databases, 13*(1), 43-72.

Levina, O. a. (2009). Realizing Event-Driven SOA. *International Conference on Internet, Web Applications and Services* (pp. 37-42). Venice: IEEE.

Li, G., Muthusamy, V., & Jacobsen, H.-A. (2010). A distributed service-oriented architecture for business process execution. *ACM Transactions on the Web, 4*(1), 2:1-2:33.

Miller, J., Palaniswami, D., Sheth, A., Kochut, K., & Singh, H. (1998). WebWork: METEOR 2's web-based workflow management system. *Journal of Intelligent Information Systems, 10*(2), 185-215.

Mühl, G., Fiege, L., & Pietzuch, P. (2006). *Distributed Event-Based Systems.* New York: Springer-Verlag.

Muth, P., Wodtke, D., Weissenfels, J., Dittrich, A., & Weikum, G. (1998). From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems, 10*(2), 159-184.

OASIS. (2007, April). *Web Services Business Process Execution Language Version 2.0*. Retrieved December 5, 2011, from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

OASIS. (2009, February). *Web Services Coordination Version 1.2*. Retrieved December 5, 2011, from http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os.doc

Reichert, M., & Bauer, T. (2007). Supporting ad-hoc changes in distributed workflow management systems. *International conference on On the move to meaningful internet systems* (pp. 150-168). Vilamoura: Springer-Verlag.

Reichert, M., & Dadam, P. (1998). Adept flex - Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems, 10*(2), 93-129.

Reichert, M., Rinderle, S., & Dadam, P. (2003). On the Common Support of Workflow Type and Instance Changes under Correctness Constraints. In R. Meersman (Ed.), *Cooperative Information Systems* (pp. 407-425). Catania: Springer.

Rinderle, S., Reichert, M., & Dadam, P. (2004). Correctness criteria for dynamic changes in workflow systems: a survey. *Data & Knowledge Engineering, 50*(1), 9-34.

Rinderle, S., Wombacher, A., & Reichert, M. (2006). Evolution of process choreographies in DYCHOR. *International Conference on Cooperative Information Systems* (pp. 273-290). Montpellier: Springer-Verlag.

Rinderle-Ma, S., & Reichert, M. (2010). Advanced Migration Strategies for Adaptive Process Management Systems. *12th Conference on Commerce and Enterprise Computing (CEC)* (pp. 56-63). Shanghai: IEEE.

van der Aalst, W. (2001). Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. *Information Systems Frontiers, 3*(3), 297-317.

van der Aalst, W., & Basten, T. (2002). Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science, 270*(1-2), 125–203.

van der Aalst, W., & ter Hofstede, A. (2005). YAWL: yet another workflow language. *Information Systems, 30*(4), 245-275.

van der Aalst, W., Pesic, M., & Schonenberg, H. (2009). Declarative workflows: Balancing between flexibility and support. *Computer Science-Research and Development, 23*(2), 99-113.

van der Aalst, W., Ter Hofstede, A., Kiepuszewski, B., & Barros, A. (2003). Workflow patterns. *Distributed and parallel databases, 14*(3), 5-51.

Weber, B., Reichert, M., & Rinderle, S. (2008). Change patterns and change support features- enhancing flexibility in process-aware information systems. *Data & knowledge engineering, 66*(3), 438-466.

Weber, B., Sadiq, S., & Reichert, M. (2009). Beyond Rigidity - Dynamic Process Lifecycle Support: A Survey on Dynamic Changes in Process-aware Information Systems. *Computer Science - Research and Development, 23*(2), 47-65.

Weske, M. (1998). Flexible modeling and execution of workflow activities. *International Conference on System Sciences* (pp. 713-722). Kohala Coast: IEEE.

Wieringa, R. (2009). Design science as nested problem solving. In S. Purao (Ed.), *4th International Conference on Design Science Research in Information Systems and Technology* (pp. 1-12). New York: ACM.

Wodtke, D., Weissenfels, J., Weikum, G., & Dittrich, A. (1996). The Mentor Project: Steps Towards Enterprise-Wide Workflow Management. *Conference on Data Engineering* (pp. 556-565). New Orleans: IEEE.

Footnotes

[1] Note that the protocol only describes incremental schema versioning, i.e. migration from v1 to v2 to v3 and so on. To allow migration of *multiple* older versions to the new version, e.g. migration from schema v1 to schema v3, steps 1 to 5 of the protocol have to be executed for each combination of running schema versions and the new to-be deployed version.

[2] Suspending a process fragment instance does not mean that the actual running activity should be suspended (e.g. suspend a boat shipment). The activity is still allowed to continue and finish, but the status of the instance will not change from ‘running’ to ‘completed’ once executed (see *Figure 8*).