Enforcing the Repeated Execution of Logic in Workflows

Mirko Sonntag, Dimka Karastoyanova
Institute of Architecture of Application Systems
University of Stuttgart, Universitaetsstrasse 38
70569 Stuttgart, Germany
{sonntag, karastoyanova}@iaas.uni-stuttgart.de

***Abstract*—The repeated execution of workflow logic is a feature needed in many situations. Repetition of activities can be modeled with workflow constructs (e.g., loops) or external workflow configurations, or can be triggered by a user action during workflow execution. While the first two options are state of the art in the workflow technology, the latter is currently insufficiently addressed in literature and practice. We argue that a manually triggered rerun operation enables both business users and scientists to react to unforeseen problems and thus improves workflow robustness, allows scientists steering the convergence of scientific results, and facilitates an explorative workflow development as required in scientific workflows. In this paper, we therefore formalize operations for the repeated enactment of activities—for both iteration and re-execution. Starting point of the rerun is an arbitrary, manually selected activity. Since we define the operations on a meta-model level, they can be implemented for different workflow languages and engines.**

Keywords-service composition; workflow adaptability; iteration; re-execution.

I. INTRODUCTION

Imperative workflow languages are used to describe all possible paths through a process. On the one hand, this ensures the exact execution of the modeled behavior without deviations. On the other hand, it is difficult, if not impossible, to react to unforeseeable and thus un-modeled situations that might happen during workflow execution, e.g., exceptions. This is the reason why flexibility features of workflows were identified as essential for the success of the technology in real world scenarios (e.g., [1]). In [2], four possible modifications of running workflows are described as advanced functions of workflow systems: the deletion of steps, the insertion of intermediary steps, the inquiry of additional information, the iteration of steps.

In this paper, we focus on the iteration of steps. Usually, iterations are explicitly modeled with loop constructs. But not all eventualities can be accounted for in a process model prior to runtime. Imagine a process with an activity to invoke a service. At runtime, the service may become unavailable. The activity and hence the process will fail, leading to a loss of time and data. Rerunning the activity (maybe with modified input parameters) could prevent this situation.

The repetition of workflow logic is not only meaningful for handling faults. In the area of scientific workflows, the result of scientific experiments or simulations is not always known a priori [3, 4]. Scientists may need to take adaptive actions during workflow execution. In this context, rerunning activities is basically useful to enforce the convergence of results, e.g., redo the generation of a Finite Element Method (FEM) grid to refine a certain area, repeat the visualization of results to obtain an image with focus on another aspect of a simulated object, or enforce the execution of an additional simulation time step.

A lot of work exists on the repetition of activities in workflows. Existing approaches use modeling constructs (e.g., BPEL retry scopes [5]), workflow configurations (e.g., Oracle BPM [6]), or automatically selected iteration start points (e.g., Pegasus [7]) to realize the repeated execution of workflow parts. An approach for the repetition of workflow logic with an arbitrary starting point that was manually selected at runtime by the user/scientist is currently missing. We argue that such functionality is useful in both business and scientific workflows. In business workflows it can help to address faulty situations, especially those where a rerun of a single faulted activity (usually a service invocation) is insufficient. In scientific workflows it is one missing puzzle piece to enable explorative workflow development [4, 8] and to control and steer the convergence of results.

In this paper, we therefore formalize two operations on workflow instances to enforce the repetition of workflow logic, namely iteration and re-execution. The *iteration* works like a loop that reruns a number of activities. The *re-execution* undoes work completed by a set of activities with the help of compensation techniques prior to the repetition of the same activities. We define the operations on the level of the workflow meta-model. Thus, the operations can be implemented in different workflow languages and engines. We discuss several problems that arise when repeating arbitrary workflow logic, such as data handling issues and the communication with clients.

The rest of the paper is organized as follows. Section II presents other work on the topic. Section III shows the workflow meta-model used in this work. Section IV describes the *iterate* and *re-execute* operations and discusses implications of the approach. Section V concludes the paper.

II. RELATED WORK

Repetition of workflow logic can be achieved language-based with certain modeling constructs. A general concept to retry and rerun transaction scopes in case of an error is shown in [9]. Eberle et al. [5] apply this concept to BPEL

 ISBN: 978-1-61208-160-1

scopes. In BPMN [10] this behavior can be modeled with sub-processes, error triggers and links. In $ADEPT_{flex}$ it is possible to model backward links to repeat faulted workflow logic [11]. In IBM MQSeries Workflow and its Flow Definition Language (FDL) activities are restarted if their exit condition evaluates to false. ADOME [12] can rerun special repeatable activities if an error occurs during activity execution. In Apache ODE an extension of BPEL invoke activities enables to retry a service invocation if a failure happens [13]. These approaches have in common that special modeling constructs realize the repetition. Thus, the iteration is pre-modeled at design time. Further, FDL and ADOME allow the rerun of a single faulted activity only. In contrast to these approaches, our solution aims at repeating a workflow starting from an arbitrary, not previously modeled point.

Iterations can also be realized by configuring workflow models with deployment information. Invoke activities in the Oracle BPEL Process Manager [6] can be configured with an external file so that service invocations are retried if a specified error occurs. The concept to retry activities until they succeed is also subject of [14]. The scientific workflow system Taverna [15] allows specifying alternate services that are taken if an activity for a service invocation fails. In contrast to these approaches, we advocate a solution where the rerun can be started spontaneously without a pre-configuration of workflows.

The scientific workflow system Pegasus can automatically re-schedule a part of a workflow if an error occurs [7]. Successfully completed tasks are not retried. Kepler's Smart Rerun Manager can be used to re-execute complete workflows [16]. Tasks that produce data that already exists are omitted. The main difference of these approaches to our idea is that we select the starting point of the iteration manually and hence can use this functionality for explorative workflow development and steering of the convergence of scientific results.

In [2] the iteration of activities is mentioned but without going into details. In the scientific workflow system e-BioFlow scientists can re-execute manually selected tasks with the help of an ad hoc workflow editor [3]. The set of activities that should be (re-)executed must be marked explicitly. No other activities are (re-)executed; no distinction is made between iteration and re-execution operations. In our approach the user only has to provide the start activity for the rerun and the successor activities are then executed as prescribed by the workflow model.

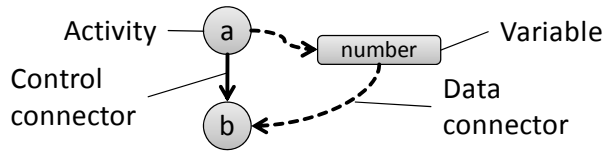

Figure 1. Example for a process model

III. META-MODEL

At first, we introduce main concepts of the workflow meta-model we use in this paper. We focus on those aspects of the meta-model needed to describe the repeated execution of workflow logic. A process model is considered a directed and acyclic graph (Figure 1). The nodes are tasks to be performed (i.e., activities). The edges are control connectors (links) and prescribe the execution order of activities. Data dependencies are represented by variables that are read and written by activities.

Definition 1 (Variable, V). The set of variables $V \subseteq M \times S$ (M = set of names; S = set of data structures) defines all variables of a process model [2]. Each $v \in V$ has assigned a finite set of possible values, its domain DOM(v) [2].

Definition 2 (Activity, A). Activities are functions that perform specific tasks. A join condition $j \in C$ with C as the set of all conditions can be assigned to an activity. If j evaluates to true at runtime, the activity is scheduled. The set of all activities of a process model is $A \subseteq M \times C$. Variables can be assigned to activities with the help of an input variable map $\iota: A \mapsto \wp(V)$ and an output variable map $o: A \mapsto \wp(V)$. Input variables may provide data to activities and activities may write data into output variables. Further, compensating activities that undo the effects of an activity can be assigned by a compensate activity map $c: N \mapsto N$.

Definition 3 (Link, L). The set $L \subseteq A \times A \times C$ denotes all control connectors in a process model. Each link connects a source with a target activity. Its transition condition $t \in C$ says if it is followed at runtime. Two activities can be connected with at most one link (i.e., links are unique).

Definition 4 (Process Model, G). A process model is a directed acyclic graph denoted by a tuple G = (m, V, A, L) with a name $m \in M$.

A. Execution And Navigation

For the execution of a process model, a new process instance of that model is created, activities are scheduled and performed, links are evaluated, and variables are read and written. These tasks (i.e., the navigation) are conducted according to certain rules. The component of a workflow system that supervises workflow execution and that implements these rules is called the navigator. We reflect the notion of time in the meta-model with ascending natural numbers. Each process instance possesses its own timeline. At time $0 \in \mathbb{N}$ a process is instantiated. Each navigation step increases the time by 1. Hence, navigation steps conducted in parallel have different time steps. In the following we present navigation rules that are most important for this work.

If an activity is executed, an activity instance is created with a new unique id. If the same activity is executed again (e.g., because it belongs to a loop), another instance of it is created with another id. The same holds for links and link instances. A new id can be generated with the function `newId()` that delivers an element of the set of ids, ID.

 ISBN: 978-1-61208-160-1

We consider process, activity and link instances sets of tuples. This allows us to navigate through a process by using set operations. Navigation steps are conducted by creating new tuples and adding them to sets (instantiation of an activity/a link) or by taking tuples from sets and adding modified tuples to sets (to change the state of existing activity/link instances).

Definition 5 (Variable Instance, $\mathcal{V}$). Variable instances provide a concrete value c for a variable v (i.e., an element of its domain) at a point in time t. The finite set of variable instances is denoted as $\mathcal{V} = \{(v, c, t) \mid v \in V, c \in DOM(v), t \in \mathbb{N}\}$. The set of all possible variable instances is $\mathcal{V}_{all}$.

Definition 6 (Activity Instance, $\mathcal{A}$). Each activity can be instantiated several times. These different instances are referred to by ids that are unique to activity instances. The set of activity instances is denoted as $\mathcal{A} = \{(id, a, s, t) \mid id \in ID, a \in A, s \in S, t \in \mathbb{N}\}$. At a point in time t an activity instance has an execution state $s \in S$ = {scheduled, executing, completed, faulted, terminated, compensated}. Note that an activity instance *a* reaches the compensated state if it is completed and its compensation activity $c(\pi_2(a))$ was executed successfully.

We define three sets that help to capture the state of a process instance and that are used to navigate through a process model graph.

Definition 7 (Active Activities, $\mathcal{R}$). The finite set of active activities $\mathcal{R}$ contains all activity instances that are scheduled or currently being executed: $\mathcal{R} \subseteq \mathcal{A}$, $\forall a \in \mathcal{R}$: $\pi_3(a) \in$ {scheduled, executing}.

Definition 8 (Finished Activities, $\mathcal{F}$). The finite set of finished activities $\mathcal{F}$ contains all activity instances that are completed, faulted or terminated: $\mathcal{F} \subseteq \mathcal{A}$, $\forall a \in \mathcal{F}$: $\pi_3(a) \in$ {completed, faulted, terminated}. Note that compensated activities are not part of $\mathcal{F}$ because their effects are undone.

Definition 9 (Active Links, $\mathcal{L}$). The finite set of active links $\mathcal{L}$ contains link instances that refer to the instantiated link e and a truth value for the evaluated transition condition: $\mathcal{L} = \{(e, t) \mid e \in E, t \in \{true, false\}\}$. $\mathcal{L}$ contains only those link instances that are evaluated but where the target activity is not yet scheduled or being executed: $\forall l \in \mathcal{L}$: $\nexists a \in \mathcal{R}$: $\pi_2(\pi_1(l)) = \pi_2(a)$.

Definition 10 (Wavefront, $\mathcal{W}$). The set of all active activities and links in a process instance is called the wavefront $\mathcal{W} = \mathcal{R} \cup \mathcal{L}$.

Definition 11 (Process Instance, p_g). An instance for a process model g is now defined as a tuple $p_g = (\mathcal{V}, \mathcal{R}, \mathcal{F}, \mathcal{L})$. The set of all process instances is denoted as $\mathcal{P}_{all}$.

As navigation example consider Figure 1. Say activity $a \in A$ is currently being executed and invokes a program that increases a given number by 1. The process instance thus looks as follows: p_g = ({(number, 100, 1)}, {(382, a, executing, 3)}, {}, {}). If activity *a* completes, its corresponding tuple is deleted from $\mathcal{R}$ and a new tuple with the new activity instance state and an increased time step is added to $\mathcal{F}$: p_g = ({(number, 100, 1)}, {}, {(382, a, completed, 4)}, {}). Now, the navigator stores the new value of the variable *number* and deletes the former value: p_g = ({(number, 101, 5)}, {}, {(382, a, completed, 4)}, {}). Even though the navigator manipulates the tuples, all these actions are recorded in the audit trail.

IV. ITERATION AND RE-EXECUTION

Based on the meta-model described above we can now address the repeated execution of workflow parts. As already proposed in [5], we also want to distinguish between two repetition operations. The first operation reruns workflow parts without taking corrective actions or undoing already completed work. The second operation resets the workflow context and execution environment with compensation techniques prior to the rerun (e.g., de-allocating reserved computing resources, undoing completed work).

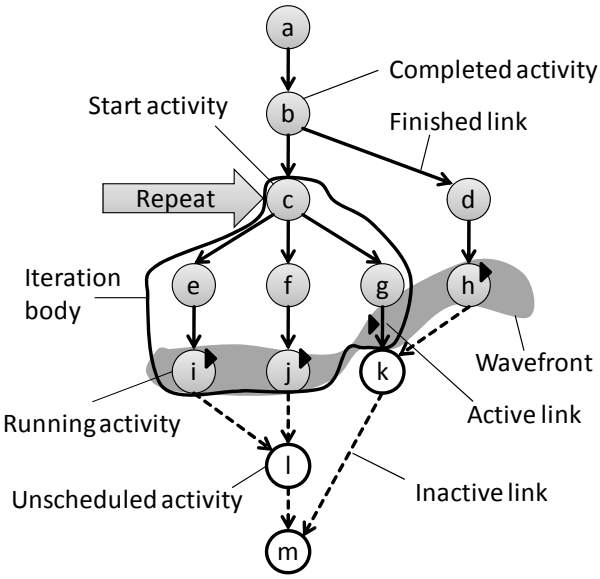

Figure 2. Example of a process instance

Before we dive into the details of the iteration of workflow parts we have to introduce several important terms (see Figure 2). The point from where a workflow part is executed repeatedly is denoted as the *start activity* (activity *c* in the figure). The start activity is chosen manually by the user/scientist at workflow runtime. The workflow logic from

the start activity to those active activities and active links that are reachable from the start activity are called *iteration body* (activities *c, e, f, g, i, j*, the links in between and link *g-k*). The iteration body is the logic that is executed repeatedly. Note that activities/links reachable from the iteration body but not in the iteration body are executed normally when the control flow reaches them (e.g., activities *k* and *l*).

For the iteration/re-execution of logic it is important to avoid race conditions, i.e., situations where two or more distinct executions of one and the same path are running in parallel. These situations can occur in cyclic workflow graphs or can be introduced by the manual rerun of activities that we propose. For example, if the repetition is started from activity *c* in Figure 2, then a race condition emerges because activities *i* and *j* on the same path are still running: activity *l* could be started if *i* and *j* complete while a competing run is started at *c*. There are two ways to avoid race conditions in this scenario. Firstly, the workflow system can wait until the running activities in the iteration body are finished without scheduling any successor activities (here: *l*). The rerun is triggered only afterwards. Secondly, running activities in the iteration body can be terminated and the rerun can start immediately. A workflow system should provide both options to the users because in some cases it is meaningful to complete running work prior to the rerun while in other cases the result of running work is unimportant. This has to be decided on a per-case-basis by the user. In the rest of this paper we focus on the more complex second option: termination. We therefore need to define an operation that terminates running activities.

Definition 12 (Termination). The terminate operation prematurely aborts running activities. Let $n = (id, a, s, t) \in \mathcal{R}$ with $s \in \{active, executing\}$ be an activity instance. Then `terminate(n)` delivers the tuple (id, a, terminated, t') with $t' > t$, i.e., the activity instance is terminated.

Definition 13 (Active Successor Activities). We need a function that finds all activities in the wavefront that belong to the iteration body. The function delivers exactly those running activities that have to be terminated before the rerun can be started:

$$\text{activeSuccActivities} : A \times \mathcal{P}_{all} \mapsto \wp(\mathcal{R})$$

Let $a \in A$ be an activity in process model g and $p_g \in \mathcal{P}_{all}$ an instance of g. Then `activeSuccActivities`$(a, p_g) = \{r_1, \ldots, r_k\}$, $r_1, \ldots, r_k \in \mathcal{R} \Leftrightarrow \forall i \in \{1, \ldots, k\}$: $\pi_2(r_i)$ is reachable from *a*.

Race conditions can also occur if active links remain in the process instance. In Figure 2, a race condition could appear as follows. If activity *k* completes and the link *h-k* is evaluated, the join condition of *k* could become true. *k* would then be started although a competing execution of the same path arises due to the repetition of *c*. That is why such links have to be found and reset.

Definition 14 (Active Successor Links). A function is needed that finds all links in the wavefront where the source activity is reachable from a given activity:

$$\text{activeSuccLinks} : A \times \mathcal{P}_{all} \mapsto \wp(\mathcal{L})$$

Let $a \in A$ be an activity of process model g and $p_g \in \mathcal{P}_{all}$ an instance of g. Then `activeSuccLinks`$(a, p_g) = \{l_1, \ldots, l_k\}$, $l_1, \ldots, l_k \in \mathcal{L} \Leftrightarrow \forall i \in \{1, \ldots, k\}$: $\pi_1(l_i)$ is reachable from *a*.

A. Iteration

Parts of a workflow may be repeated without the need to undo any formerly completed work. A scientist may want to enforce the convergence of experiment results and therefore repeats some steps of a scientific workflow. This is what we denote as iteration of workflow parts.

Definition 15 (Iterate Operation). The iteration is a function that repeats logic of a process model for a given process instance beginning with the given activity as starting point and taking the data indicated by the given time step as input for the next iteration.

$$\mathfrak{i} : A \times \mathcal{P}_{all} \times \mathbb{N} \mapsto \mathcal{P}_{all}$$

Let $a \in A$ be the start activity of the iteration and $p\_in_g, p\_out_g \in \mathcal{P}_{all}$ two process instances. Here, p_in_g is the input for the $\mathfrak{i}$ operation and p_out_g is the resulting instance with changed state that is ready to start with the iteration. As pre-condition we define that only already executed activities can be used as start activity: $\exists n \in \mathcal{R} \cup \mathcal{F} : \pi_2(n) = a$. This prevents (1) using the operation on dead paths, (2) jumping into the future of a process instance, and (3) guarantees the correct termination of running activities. Then $\mathfrak{i}(a, p\_in_g, t) = p\_out_g$ with $t \in \mathbb{N}$, $p\_in_g = (\mathcal{V}_{in}, \mathcal{R}_{in}, \mathcal{F}_{in}, \mathcal{L}_{in})$ and $p\_out_g = (\mathcal{V}_{out}, \mathcal{R}_{out}, \mathcal{F}_{out}, \mathcal{L}_{out}) : \Leftrightarrow$

1. $\mathcal{V}_{out} = \mathcal{V}_{in}$
2. $\mathcal{R}_{out} = \mathcal{R}_{in} \setminus \text{activeSuccActivities}(a) \cup \{$(`newId()`, *a*, active, *i*)$\}$, *i* is a new and youngest time step
3. $\mathcal{F}_{out} = \mathcal{F}_{in} \cup \cup_{n \in \text{activeSuccActivities}(a)} \text{terminate}(n)$
4. $\mathcal{L}_2 = \mathcal{L}_1 \setminus \text{activeSuccLinks}(a)$

The variables remain unchanged (1.). All active successor activities from *a* are terminated, i.e., deleted from the set of running activities (2.) and inserted with a new status to the set of finished activities (3.). All active links in the iteration body are reset (4.). The start activity is scheduled (added to the set of active activities with status *active*) so that the workflow logic is repeated beginning with the start activity (2.). The join condition of the start activity is not evaluated again.

B. Re-execution

It is also needed to repeat parts of a workflow as if they were executed for the first time. Completed work in the iteration body has to be reversed/compensated prior to the repetition. A scientist may want to retry a part of an experiment because something went wrong. But the

 ISBN: 978-1-61208-160-1

execution environment has to be reset first. This is what we denote as re-execution of workflow parts.

Algorithm 1 (Compensate Iteration Body). For the compensation of completed work in the iteration body we propose an algorithm with the following signature:

compensateIterationBody : A $\times$ $\mathcal{P}_{all} \mapsto \wp(\mathcal{V}_{all})$

The function compensates all completed activities of the iteration body in reverse execution order. It delivers the values of variables that were changed during compensation. Let $a \in$ A be the start activity of the re-execution and $p \in \mathcal{P}_{all}$ a process instance for the model of *a*. Then compensateIterationBody(*a*, *p*) = {v_1, …, v_k} with $p = (\mathcal{V}, \mathcal{R}, \mathcal{F}, \mathcal{L})$, v_1, … , $v_k \in \mathcal{V}_{all}$ works as follows (Note: f.state for f $\in \mathcal{F}$ is equivalent to π_3(f); f.time to π_4(f)):

```
function compensateIterationBody(a, p)
1  V_result ← ∅
2  F = {f ∈ F | f.state == completed ∧
   π2(f) is reachable from a}
3  while (|F| > 0) do
4   if |F| > 1 then
5    ∃m ∈ F: ∀n ∈ F, n ≠ m:
     m.time > n.time ⇒ execute
     compensating activity c(π2(m))
6   else
7    ∃m ∈ F ⇒ execute compensating
     activity c(π2(m))
8   F ← F \ {m}
9   ∀v ∈ o(c(π2(m))): V_result ← V_result ∪ {(v,
    c, t)}, c is the new value of variable
    v, t is the timestamp of the
    assignment
10 od
11 return V_result
```

A similar algorithm for the creation of the reverse order graph is also proposed in [17]. But the intention of our algorithm is to deliver the changed variable values as result of the compensation operation.

Definition 17 (Re-execute Operation). The re-execution is a function that repeats logic of a process model for a given process instance with a given activity as starting point. The data indicated by the given time step is taken as input for the re-execution. The operation uses the compensate operation for already completed work in the iteration body.

$\mathfrak{r}$: A $\times$ $\mathcal{P}_{all}$ $\times$ $\mathbb{N} \mapsto \mathcal{P}_{all}$

$a \in$ A, p_in_g, $p\_out_g \in \mathcal{P}_{all}$ and the pre-condition are similar to the iterate operation. The difference is the calculation of $\mathcal{V}_{out}$: $\mathfrak{r}(a, p\_in_g, t) = p\_out_g$: $\Leftrightarrow$

1. $\mathcal{V}_{out} = \mathcal{V}_{in} \cup$ compensateIterationBody(*a*, *p*)

The variable values might be modified as a result of the compensation of completed work in the iteration body (1.). Note that the start activity for the re-execution is scheduled after the compensation is done.

C. Data Handling

For the repetition of workflow parts the handling of data is of utmost importance. Where to store data that the former iteration has produced? What data should be taken as input for the next iteration? A mechanism is needed to store different values for same variables and to load variable values for iterations. The compensation of completed work as is done in the re-execution operation is not sufficient for resetting variables because compensation does not always set the variables to their former states. This strongly depends on the compensation logic and invoked services.

The desired functionality can be realized by saving the complete content history of variables including the assignment timestamps. Many workflow systems already provide this as part of their audit trail [2, 18]. If a variable is changed a new tuple is inserted in $\mathcal{V}$ and the former tuples remain, e.g., at time step 9 variable *number* was increased by 1: $\mathcal{V}$ = {(number, 50, 1), (number, 51, 9)}. That way no data is lost due to repetition of workflows parts and former variable values can be accessed as input for the rerun. It must be possible for the users to choose the input for the next iteration. That is why we foresee the specification of a time step in the iteration and re-execution operation (see Definition 15 and 17). This time step indicates which variable values are taken as input for the respective operation, namely those that were valid at the given point in time. The visible variables have to be re-initialized accordingly. In Figure 3, the sample workflow of Figure 1 was iterated from activity *a* three times leading to a chain of executions of activities *a* and *b*. The current value of variable *number* was taken for each rerun. Imagine the user wants to iterate again from activity *a*. Different time steps chosen by the user as input for the operation would influence the initialization of variables for the iteration as follows. At the (latest) time t = 8 the value of *number* is 102; at t = 6 the value is the one obtained after the second execution of *a* (101); and at t = 1 *number* has its initial value (99).

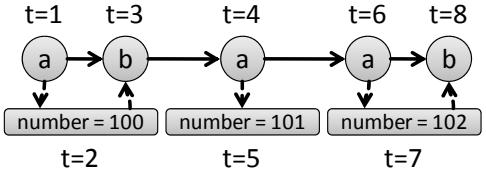

Figure 3. Data handling during workflow repetition

D. Implications on the Execution Correctness

In practice, workflows consist of different activity types, e.g., for sending/receiving messages, loops. The enforced repetition of workflow logic has to account for different activity types, especially those that interact with external

entities such as clients, humans, services/programs. The main problem is that the repetitions are not reflected in the workflow logic and hence the aforementioned entities do not know a priori the exact behavior of the workflow.

If a message receiving activity is repeated, the message sending client has to re-send the message or send an adapted message. The problem is that the client needs to be informed about the repetition. A simple solution is that clients provide a special operation that can be used by the workflow engine to propagate the iteration. Over this operation the engine could send the message back to the client enriched with further context (e.g., the address of the engine, correlation information, workflow instance id). The client then decides whether to re-send the message or to send an adapted one.

The repetition of message sending activities is straight forward for idempotent services. Non-idempotent services should be compensated prior to a repeated invocation, as is done in the re-execution operation. If the iteration operation repeats the execution of non-idempotent services, then the user is responsible for the effect of the operation.

Iterations within modeled loops can have an unforeseeable impact on the behavior of workflows. The context of workflows might be changed in a way that leads to infinite loops (e.g., because the repetition changes variable values so that a while condition can never evaluate to false). Usually, a workflow system provides operations to change variable values. This functionality can be used to resolve infinite loops.

E. User Interaction With the Workflow System

A workflow system that implements our approach must provide a monitoring tool that allows users to continuously follow the execution state of process instances. If the user detects a faulty or unintended situation, he can suspend the workflow and manually trigger an iteration/re-execution. The system requests him for the time step used to retrieve the variable values for the loop. Then, the process instance state is changed as described in Section IV and the user can resume workflow execution.

V. CONCLUSION AND OUTLOOK

In this paper, we have formally described two operations to enforce the repetition of workflow logic during workflow runtime: the *iterate* operation reruns activities starting from a manually selected activity; the *re-execute* operation undoes completed work in the iteration body before rerunning activities. The distinctive features of the approach are that the repetition does not have to be modeled or configured previously and that arbitrary activities can be used as starting point for the rerun. We have shown how problems with the data handling and communication with external parties can be solved. The approach is described based on an abstract meta-model and thus can be applied to existing or future workflow engines and languages. Currently, we are working on an implementation for the BPEL engine Apache ODE.

The enforced repetition of workflow logic is a step towards our goal to enable an explorative workflow development, especially in the field of scientific workflows.

ACKNOWLEDGMENT

The authors would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/1) at the University of Stuttgart.

REFERENCES

[1] W. van der Aalst, T. Basten, H. Verbeek, P. Verkoulen, and M. Voorhoeve, “Adaptive workflow: on the interplay between flexibility and support,” Proc. of the 1st Conf. on Enterprise Information Systems, pp. 353-360, 1999.

[2] F. Leymann and D. Roller, “Production Workflow – Concepts and Techniques,” Prentice Hall, 2000.

[3] I. Wassink, M. Ooms, and P. van der Vet, “Designing workflows on the fly using e-BioFlow,“ ICSOC, 2009.

[4] R. Barga and D. Gannon, “Scientific vs. business workflows,” in: Taylor et al., “Workflows for e-Science,” Springer, pp. 9-18, 2007.

[5] H. Eberle, O. Kopp, F. Leymann, and T. Unger, “Retry scopes to enable robust workflow execution in pervasive environments,” Proc. of the 2nd MONA+ Workshop, 2009.

[6] Oracle BPEL Process Manager, http://www.oracle.com/us/products/middleware/application-server/bpel-home-066588.html

[7] E. Deelman, G. Mehta, G. Singh, M.-H. Su, and K. Vahi, “Pegasus: Mapping large-scale workflows to distributed ressources,” In: Taylor et al., “Workflows for e-science,” Springer, pp. 376-394, 2007.

[8] G. Vossen and M. Weske, “The WASA approach to workflow management for scientific applications,” Workflow Management Systems and Interoperability, NATO ASI Series F: Computer and System Sciences, Vol. 164, Springer-Verlag, pp. 145-164, 1998

[9] F. Leymann, “Supporting business transactions via partial backward recovery in workflow management systems,” Proc. of the BTW, Springer, 1995.

[10] Object Management Group (OMG), “Business Process Modeling Notation (BPMN) Version 1.2,” OMG Specification, 2009.

[11] M. Reichert and P. Dadam, “$ADEPT_{flex}$ – Supporting dynamic changes of workflows without losing control,“ Intelligent Information Systems, vol. 10, pp. 93-129, 1998.

[12] D. Chiu, Q. Li, and K. Karlapalem, “A meta modeling approach to workflow management systems supporting exception handling,” Information Systems, vol. 24, pp. 159-184, 1999.

[13] Apache ODE, http://ode.apache.org/activity-failure-and-recovery.html

[14] P. Greenfield, A. Fekete, J. Jang, D. Kuo, “Compensation is not enough,” Proc. of the 7th EDOC, 2003.

[15] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, “Taverna: a tool for building and running workflows of services,” Nucleic Acids Research, vol. 34, Web Server issue, pp. 729-732, 2006.

[16] I. Altintas, O. Barney, E. Jaeger-Frank, “Provenance Collection Support in the Kepler Scientific Workflow System,“ Provenance and Annotation of Data, IPAW, LNCS, Vol. 4145, pp. 118–132, Springer, 2006.

[17] R. Khalaf, “Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective,” Doctoral Thesis, ISBN: 978-3-86624-344-6, 2008.

[18] Workflow Management Coalition, “Audit Data Specification, Version 1.1,” WfMC Specification, 1998.

 ISBN: 978-1-61208-160-1