# Ad hoc Iteration and Re-execution of Activities in Workflows

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

*Abstract*—**The repeated execution of workflow logic is usually modeled with loop constructs in the workflow model. But there are cases where it is not known at design time that a subset of activities has to be rerun during workflow execution. For instance in e-Science, scientists might have to spontaneously repeat a part of an experiment modeled and executed as workflow in order to gain meaningful results. In general, a manually triggered ad hoc rerun enables users reacting to unforeseen problems and thus improves workflow robustness. It allows natural scientists steering the convergence of scientific results, business analysts controlling their analyses results, and it facilitates an explorative workflow development as required in scientific workflows. In this paper, two operations are formalized for a manually enforced repeated enactment of activities, the iteration and the re-execution. The focus thereby lies on an arbitrary, user-selected activity as a starting point of the rerun. Important topics discussed in this context are handling of data, rerun of activities in activity sequences as well as in parallel and alternative branches, implications on the communication with partners/services and the application of the concept to workflow languages with hierarchically nested activities. Since the operations are defined on a meta-model level, they can be implemented for different workflow languages and engines.**

***Keywords-workflow ad hoc adaptation; iteration; re-execution; service composition***

## 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/or un-modeled situations that might happen during workflow execution, e.g., exceptions, changes in regulations in business processes, etc. This is the reason why flexibility features of workflows were identified as essential for the success of the technology in real world scenarios [2, 3, 4]. In [5], 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.

This paper focusses on the iteration of steps. Usually, iterations are explicitly modeled with loop constructs. However, 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, if the underlying service middleware cannot tackle the problem with failing services. An ad hoc operation to rerun the activity (maybe with modified input parameters) could prevent this situation.

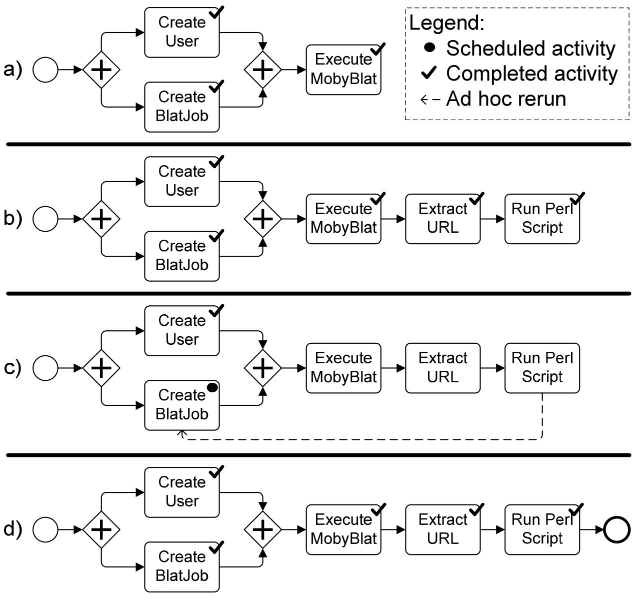


Figure 1. Example for the flexible development of a scientific workflow (borrowed from [6])

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 [6, 7, 8, 9]. 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 in the grid, repeat the visualization of results to obtain an image with focus on another aspect of a simulated object, enforce the execution of an additional simulation time step. A simplified example for an explorative development of a scientific workflow is given in Figure 1 (the example is borrowed from [6]). In this scenario, a scientist wants to perform a search for a DNA sequence in a particular genome using a Blat Web service. He models a workflow with three tasks and puts them in the order presented in (Figure 1a): "Execute MobyBlat" invokes the scientific Web service; "Create User" creates the input for MobyBlat containing a specific database to search in and providing user credentials; "Create BlatJob" configures the search operation and contains the DNA sequence to search for in the selected genome. The scientist runs this workflow (Figure 1a). He takes a look at the result of the MobyBlat service and discovers that the result format is a MOBY-S XML object. The result object contains a URL to the final result, the Blat report. In order to download the report he adapts the running workflow by appending two additional tasks: "Extract URL" gets the URL to the Blat report out of the MOBY-S XML object; "Run Perl Script" starts a Perl script that downloads the report (Figure 1b). The scientist inspects the downloaded report and recognizes that it has an inappropriate format. Hence, he reruns the workflow from the "Create BlatJob" task on (Figure 1c). In this second execution, he configures the BlatJob so that the Blat Web service delivers the expected format (Figure 1d). With this the scientist finishes the development of this scientific workflow in an iterative manner. The ad hoc adaptation of the workflow and the ad hoc rerun operation prevent a loss of data, time and money compared to a restart of the complete workflow and hence the creation of a new workflow instance. This is especially the case for long-running (scientific) workflows. In the example, the scientist does not have to provide the input for the "Create User" task again. There are other scenarios where the visualization of scientific results is repeated several times with different parameters without a need to rerun the complete long-running scientific simulation.

A significant number of approaches exist for enabling the repetition of activities in workflows. Existing approaches use modeling constructs (e.g., loops, BPEL retry scopes [10]), workflow configurations (e.g., Oracle BPM [11]), or an automatic rerun of faulted activities (e.g., Pegasus [12]) to realize the repeated execution of workflow parts. An approach for the ad hoc repetition of workflow logic with an *arbitrary starting point* that was user-selected at runtime is currently missing in industrial workflow engines and insufficiently addressed in research. 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, or changes in the control logic needed to address new requirements. In scientific workflows it is one missing puzzle piece to enable explorative workflow development [7, 8] and to control and steer the convergence of results.

This paper therefore focusses on enabling the rerun of activities in workflows from arbitrary points in the workflow model. Two operations on workflow instances are formalized to enforce the repetition of workflow logic: 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. The operations are defined on the level of the workflow meta-model. Thus, the operations can be implemented in different workflow languages and engines. Problems such as data handling issues, the communication with partners, or how the concept can be applied in workflow languages with block structures are identified and discussed. This paper is a logical continuation of our work presented in [1]. Note that the terms "workflow" and "process" are used interchangeably.

The rest of the paper is organized as follows. Section II shows the workflow meta-model used in this work. Section III describes the *iterate* and *re-execute* operations. Section IV addresses data handling issues and Section V applies the approach in more complex workflow graphs including parallel and alternative branches. Section VI discusses implications of the approach on message-receiving and message-sending activities, on reruns within loops, and on reruns in workflow languages with block structures. Section VII shows how users interact with a workflow system that implements the manually enforced repetition of workflow logic. Section VIII is devoted to the prototypical implementation of the concepts based on BPEL. Section IX presents work related to the research topic of this paper. Finally, Section X concludes the paper.

## II. META-MODEL

The workflow meta-model used in this paper is based on the one provided in [5]. It is adapted where appropriate in order to accommodate the aspects needed to describe the repeated execution of workflow logic. A process model is considered a directed, acyclic graph (Figure 2). The nodes are tasks to be performed (i.e., activities). The edges are control connectors (or links) and prescribe the execution order of activities. Data dependencies are represented by variables that are read and written by activities. In the description of the meta-model $\wp(S)$ is used as the power set of a given set S.

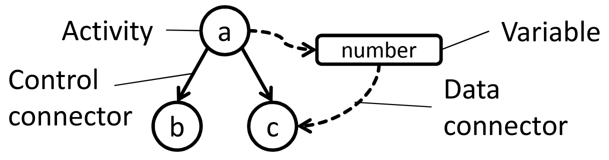


Figure 2. Example for a process model

### A. Modeling

A workflow model can be expressed with the help of sets for the different workflow elements defined in the following.

**Definition "Variables, V".** The set of variables defines all variables of a process model:

$$V \subseteq M \times S \tag{1}$$

M is the set of names and S denotes the set of data structures. Each $v \in V$ has assigned a finite set of possible values, its domain DOM(v) [5].

**Definition "Activities, A".** Activities are functions that perform tasks. The set of all activities of a process model is

$$A \subseteq M \times C. \tag{2}$$

C is the set of all conditions in a process model and is used here as join condition for an activity. If $j \in C$ evaluates to `true` at runtime, the activity is instantiated and scheduled (i.e. the navigator is going to execute the activity). Variables can be assigned to activities via an input variable map

$$i: A \mapsto \wp(V) \tag{3}$$

and an output variable map

$$o: A \mapsto \wp(V). \tag{4}$$

Input variables may provide data to activities and activities may write data into output variables. Furthermore, compensating activities that undo the effects of an activity can be assigned by a compensate activity map

$$c: A \mapsto A. \tag{5}$$

This map reflects the concept that activities can be considered as pairs consisting of an activity and its compensating activity. The idea is geared towards the approach of sagas [13]: The workflow can thereby be considered as a long-lived transaction implemented as saga, i.e. as non-atomic transaction that consists of a sequence of atomic sub-transactions $T_1, \ldots, T_n$; an activity $a \in A$ with a compensating activity is like an atomic sub-transaction $T_j$ in a saga, and the compensating activity c(a) can be compared to a compensating transaction $C_j$.

**Definition "Links, L".** The set that denotes all control connectors/links in a process model is

$$L \subseteq A \times A \times C. \tag{6}$$

Each link connects a source with a target activity. Its transition condition $t \in C$ determines at runtime if the link is followed. Two activities can be connected with at most one link (i.e., links are unique).

**Definition "Process Model, G".** A process model is a directed acyclic graph denoted by a tuple

$$G = (m, V, A, L) \tag{7}$$

with a name $m \in M$.

## B. *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*. The notion of time in the meta-model is reflected 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. In the following, the navigation rules that are most important for this work are presented.

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 activity instance 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.

Process, activity and link instances are considered sets of tuples. This allows navigating 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 deleting tuples from sets and adding modified tuples (to change the state of existing activity/link instances).

**Definition "Variable Instances, $V^I$".** 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

$$V^I = \{(v, c, t) \mid v \in V, c \in DOM(v), t \in \mathbb{N}\}. \tag{8}$$

The set of all possible variable instances is $V^I_{all}$.

**Definition "Activity Instances, $A^I$".** The set of activity instances is denoted as

$$A^I = \{(id, a, s, t) \mid id \in ID, a \in A, s \in S, t \in \mathbb{N}\}. \tag{9}$$

At a point in time t an activity instance $a \in A^I$ has an execution state $s \in S = \{$`S`, `E`, `C`, `F`, `T`, `COMP`, `D`$\}$. The meaning of the states is as follows:

- `S`, scheduled: The activity is in the execution queue of the navigator but not yet running. The navigator is going to execute the activity in future.
- `E`, executing: The activity is running.
- `C`, completed: The activity was successfully executed.
- `F`, faulted: A fault happened during activity execution.
- `T`, terminated: Abortion of a scheduled or executing activity by the user.
- `COMP`, compensated: The compensation activity c(`model`(*a*)) was executed successfully for a completed activity.
- `D`, dead: The activity is located in a dead path, i.e., a path with links evaluated to `false`. It was neither scheduled nor executed.

The function `model`(*a*) for an activity instance *a* = (id, a, s, t) $\in A^I$ delivers its activity model a. Note that there is at most one instance of an activity in $A^I$. That way $A^I$ exactly reflects the process instance state in the current iteration. There is no influence by activity states from former iterations. While this condition is inherent for workflows without loops, it must be explicitly ensured by the navigator component of the workflow engine for more complex workflow executions including loops or manual ad hoc reruns of activities (in the focus of this work).

In the following, three sets are defined that help to capture the state of a process instance and that are used to navigate through a process model graph. These sets extend the meta-model described in [5].

**Definition "Active Activities, $A^A$".** The finite set of active activities $A^A$ contains all activity instances that are scheduled or currently being executed:

$$A^A \subseteq A^I, \forall a \in A^A\text{: } \texttt{state}(a) \in \{\texttt{S}, \texttt{E}\}. \tag{10}$$

The function $\texttt{state}(a)$ for an activity instance $a$ = (id, a, s, t) $\in A^I$ returns its current state s $\in$ S.

**Definition "Finished Activities, $A^F$".** The finite set of finished activities $A^F$ contains all activity instances that are completed, faulted, terminated, or dead:

$$A^F \subseteq A^I, \forall a \in A^F\text{: } \texttt{state}(a) \in \{\texttt{C}, \texttt{F}, \texttt{T}, \texttt{D}\}. \tag{11}$$

This set is needed to assure a preconditions for the repetition of activities and for the compensation of already completed work. Note that compensated activities are not part of $A^F$ because their effects are undone.

**Definition "Evaluated Links, $L^E$".** The finite set of evaluated links $L^E$ contains link instances whose transition condition is already interpreted. Link instances refer to the instantiated link l, have a truth value c for the evaluated transition condition and an execution time t:

$$L^E = \{(l, c, t) \mid l \in L, c \in \{\texttt{true}, \texttt{false}\}, t \in \mathbb{N}\}. \tag{12}$$

Note that each link has at most one link instance in $L^E$ for one process instance. If a link is evaluated repeatedly (e.g., due to a loop or a manual ad hoc rerun), the old link instance must be removed from $L^E$. This is ensured by the navigator component of the workflow engine in order to prevent an interference of link instances of different workflow iterations. Note that the set of evaluated links is usually not part of the context of a workflow instance in typical workflow engines (cf. [5, 14, 15]). The link state (i.e., the truth value c) is only important to evaluate the join condition of the link's target activity and can be thrown away afterwards. In this work, the context of process instances is extended by storing the truth value for all evaluated links because it is needed for a correct join behavior if join activities are rerun. The set $L^E$ is very similar to the markings of control connectors known from ADEPT [3, 16].

**Definition "Wavefront, W".** The set of all active activities and evaluated links, for which the target activity is not yet scheduled, is called the process instance's wavefront

$$W = A^A \cup L^A \tag{13}$$

with $L^A \subseteq L^E$, $\forall l \in L^A$: $\nexists a \in A^A \cup A^F$: $\texttt{target}(\texttt{model}(l)) = \texttt{model}(a)$. The function $\texttt{model}(l)$ for a link instance $l$ = (l, c, t) $\in L^E$ delivers its link model l. The function $\texttt{target}$(l) for a link l = (a, b, c) $\in$ L returns its target activity b.

**Definition "Process Instance, $p_g$".** An instance for a process model g is now defined as a tuple

$$p_g = (V^I, A^A, A^F, L^E). \tag{14}$$

TABLE I. THE NAVIGATION EXAMPLE SHOWS HOW THE WORKFLOW ENGINE EXECUTES A WORKFLOW INSTANCE BY SET OPERATIONS.

| Time | $V^I$ | $A^A$ | $A^F$ | $L^E$ |
|---|---|---|---|---|
| 1 | {(number, 100, 1)} | | {} | {} |
| 2 | {(number, 100, 1)} | {(382, a, S, 2)} | {} | {} |
| 3 | {(number, 100, 1)} | {(382, a, E, 3)} | {} | {} |
| 4 | {(number, 100, 1) , (number, 101, 4)} | {(382, a, E, 3) | {} | {} |
| 5 | {(number, 100, 1) , (number, 101, 4)} | {} | {(382, a, C, 5)} | {} |
| 6 | {(number, 100, 1) , (number, 101, 4)} | {} | {(382, a, C, 5)} | {(a-b, true, 6)} |
| 7 | {(number, 100, 1) , (number, 101, 4)} | {} | {(382, a, C, 5)} | {(a-b, true, 6), (a-c, false, 7)} |
| 8 | {(number, 100, 1) , (number, 101, 4)} | {(383, b, S, 8)} | {(382, a, C, 5)} | {(a-b, true, 6), (a-c, false, 7)} |
| 9 | {(number, 100, 1) , (number, 101, 4)} | {(383, b, E, 9)} | {(382, a, C, 5)} | {(a-b, true, 6), (a-c, false, 7)} |
| 10 | … | … | … | … |

The set of all process instances is denoted as $P_{\text{all}}$. As navigation example consider the workflow model in Figure 2 and a corresponding workflow instance in Table 1. Say the workflow is instantiated and the variable number $\in$ V is initialized with 100 (time step 1). Then, activity a $\in$ A is scheduled (2) and executed (3). Suppose activity a models the invocation of a program that increases a given number by 1. The variable "number" is used as input value for this operation and is hence updated (4), i.e., the tuple representing the variable instance is substituted. Activity a completes and its corresponding instance tuple is deleted from $A^A$ and a new tuple containing the new activity instance state with increased time step is added to $A^F$ (5). Now the navigator evaluates the transition condition of the links a-b and a-c; a-b's condition evaluates to `true` (6), a-c's to `false` (7). As a consequence, the target activity of a-b is scheduled and executed (8 and 9). Note that even though the navigator manipulates the tuples, all these actions are recorded in the audit trail [5].

## III. Iteration and Re-execution

Based on the meta-model described above the repeated execution of workflow parts is described in this section. As already proposed in [10], two repetition operations are thereby distinguished. The first operation, iteration, reruns workflow parts without taking corrective actions or undoing already completed work. The second operation, re-execution, resets the workflow context and execution environment with compensation techniques prior to the rerun (e.g., de-allocating reserved computing resources, undoing completed work).

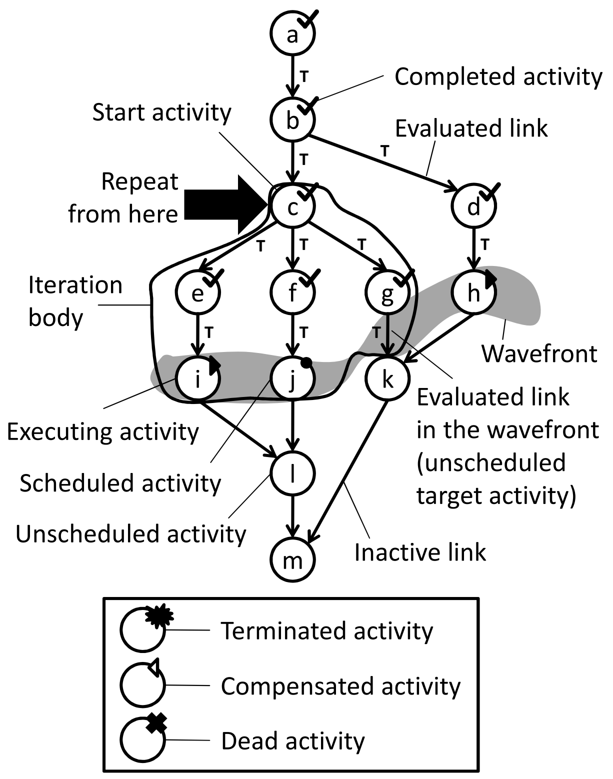


Figure 3. Example of a process instance

Before going into the details of the iteration of workflow parts several important terms are introduced (see Figure 3). 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 this work deals with. For example, if the repetition is started from activity c in Figure 3, 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 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. In some cases it is meaningful to complete running work prior to the rerun (e.g., to reach a consistent system state), in other cases an abortion is a better choice (e.g., because the result of running work is unimportant or the activities being executed are long-running). This has to be decided on a per-case-basis by the user. In the rest of the paper the focus lies on the option "termination" since it is more complex and requires one step more than the option "wait for completion". However, "wait for completion" can be derived from the descriptions by omitting the explicit termination of activities in the examples.

**Definition "Activities in Iteration Body".** A function is needed that finds all activity instances in the iteration body of an activity in a given process instance. The function is useful for terminating active activities in the iteration body (or for waiting for their completion) to avoid race conditions and for resetting finished activities to avoid interference of activity execution states in different activity runs:

$$\text{activitiesInIterationBody: A} \times P_{\text{all}} \mapsto \wp(\text{A}^{\text{I}}) \qquad (15)$$

Let a $\in$ A be an activity in process model g and $p_g \in P_{\text{all}}$ an instance of g. Then `activitiesInIterationBody`(a, $p_g$) = $\{a_1, \ldots, a_k\}$, $a_1, \ldots, a_k \in \text{A}^{\text{I}} \Leftrightarrow \forall \text{i} \in \{1, \ldots, \text{k}\}$: `model`($a_i$) is reachable from activity a. An algorithm for the "activities in iteration body" function can be implemented by walking through the workflow graph beginning with activity a until the wavefront or an already visited activity is reached. The activity instance for each considered activity is stored. Since each activity is visited at most once, the algorithm is in *O*(n), with n as the number of activities in the workflow model.

Race conditions can also occur if evaluated links in the iteration body remain in the process instance. In Figure 3, a race condition could appear as follows. If activity h completes and the link h-k is evaluated, the join condition of activity k could become true. Activity k would then be started although a competing execution of the same path arises due to the repetition of activity c. That is why such links have to be found and reset, i.e., they are deleted from the set of evaluated links $L^E$.

**Definition "Links in Iteration Body".** A function is needed that finds all evaluated links in the iteration body in a given activity and process instance:

$$\text{linksInIterationBody: } A \times P_{\text{all}} \mapsto \wp(L^E) \quad (16)$$

Let $a \in A$ be an activity of process model g and $p_g \in P_{\text{all}}$ an instance of g. Then $\texttt{linksInIterationBody}(a, p_g) = \{l_1, \ldots, l_k\}, l_1, \ldots, l_k \in L^E \Leftrightarrow \forall i \in \{1, \ldots, k\}\text{: } \texttt{model}(l_i)$ is reachable from activity a. An algorithm for the "links in iteration body" function can be implemented by traversing the workflow graph starting from activity a. Each path has to be followed only until the wavefront or an already visited activity is reached. Since each link is visited at most once, the complexity of such an algorithm is in $O(n)$, where n is the number of activities in the workflow model.

## 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.

**Definition "Iterate Operation".** The iteration is a function that repeats logic of a process model for a given process instance. A specified activity is the starting point of the operation. The input data elements for the iteration are either the current variable values or are loaded from a specified variable snapshot that belongs to the start activity.

$$\iota\text{: } A \times P_{\text{all}} \mapsto P_{\text{all}} \quad (17)$$

Let $a \in A$ be the start activity of the iteration and $p\_in_g, p\_out_g \in P_{\text{all}}$ two process instances. Here, $p\_in_g$ is the input for the ι operation and $p\_out_g$ is the resulting instance with changed state that is ready to start with the iteration. The pre-condition is that only already instantiated but no dead activities can be used as start activity:

$$\exists n \in A^A \cup A^F\text{: } \texttt{state}(n) \notin \{\texttt{D}\} \land \texttt{model}(n) = a. \quad (18)$$

This prevents (1) using the operation on dead paths and (2) jumping into the future of a process instance, which are both not a repetition of completed workflow logic.

Then $\iota(a, p\_in_g) = p\_out_g$, $p\_in_g = (V^I_{in}, A^A_{in}, A^F_{in}, L^E_{in})$ and $p\_out_g = (V^I_{out}, A^A_{out}, A^F_{out}, L^E_{out}) :\Leftrightarrow$

1. $V^I_{out} = V^I_{in}$
2. $A^A_{out} = A^A_{in} \setminus \text{activitiesInIterationBody}(a, p\_in_g) \cup \{(\texttt{newId()}, a, \texttt{S}, t)\}$, t is a new and youngest time step
3. $A^F_{out} = A^F_{in} \setminus \text{activitiesInIterationBody}(a, p\_in_g)$
4. $L^E_{out} = L^E_{in} \setminus \text{linksInIterationBody}(a, p\_in_g)$

The variables remain unchanged (1.). This reflects the case where the current variable values are taken as input for the iteration. All active successor activities from a are terminated, i.e., deleted from the set of running activities $A^A$ (2.). All finished activities in the iteration body are reset, i.e., removed from the set of finished activities $A^F$ (3.). All evaluated links in the iteration body are reset, i.e., their evaluation result is deleted from the set of evaluated links $L^E$ (4.). The start activity is scheduled (added to the set of active activities with status scheduled, `S`) so that the workflow logic is repeated beginning with the start activity (2.). The join condition of the start activity is not evaluated again.

In the second case of the ι operation, a variable snapshot is loaded prior to the iteration. The loaded variable values are taken as input for the iteration:

1. $V^I_{out} = V^I_{in} \cup \texttt{loadSnapshot}(b, p\_in_g, e, V')$

Here, $b \in A$ is the activity to load the snapshot for (the start activity or a predecessor thereof), $e \in \mathbb{N}$ is the execution number of b needed to select the correct snapshot instance, $V' \subseteq V$ is a subset of variables to be loaded from the snapshot. The complete definition of the function can be found in Section IV in (21). Shapshots are stored during process execution before each activity that modifies variables. A snapshot is uniquely addressed by its corresponding activity b and an execution number e. The latter is needed because there can be several snapshot instances for an activity—one for each activity execution. The subset of variables V' can be specified by the user to select particular variables that should be loaded from a snapshot. That means it is possible to load only a part of a snapshot. This is especially important for iterations in parallel paths. Variables that are not loaded from the snapshot have the same value as in the original process instance $p\_in_g$. For more details about data handling see Section IV.

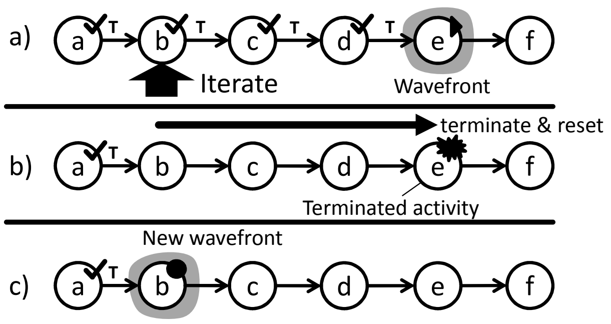


Figure 4. Iteration in a sequence of activities. The user requests the iteration of activities (a). The iteration body is reset and active activities are terminated (b). Finally, the start activity of the iteration is scheduled (c).

As an example for the ad hoc iteration of workflow logic consider Figure 4. There is a sequence of activities. Activity e is currently being executed. The user wants to iterate the workflow with activity b as start activity (Figure 4a). The path from b to the wavefront is traversed, visited links are reset (b-c, c-d and d-e in the example), and scheduled or running activities are terminated (activity e), as shown in Figure 4b. Finally, a data snapshot is loaded (if requested by the user) and the start activity (b) is scheduled (Figure 4c).

## 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 execution environment has to be reset first (e.g. stateful services have to be set to their initial state, computing resources have to be released).

**Algorithm "Compensate Iteration Body".** For the compensation of completed work in the iteration body an algorithm with the following signature is defined:

$$\text{compensateIterationBody: A} \times P_{\text{all}} \mapsto \wp(\text{V}^{\text{I}}_{\text{all}}) \qquad (19)$$

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 $p \in P_{\text{all}}$ a process instance for the model of activity a. Then `compensateIterationBody`(a, $p$) = $\{v_1, \ldots, v_k\}$ with $p = (\text{V}^{\text{I}}, \text{A}^{\text{A}}, \text{A}^{\text{F}}, \text{L}^{\text{E}})$, $v_1, \ldots, v_k \in \text{V}^{\text{I}}_{\text{all}}$ works as follows (Note: The function `time`(f) with $f \in \text{A}^{\text{I}}$ delivers the time of the last state change of activity instance f.):

```
function compensateIterationBody(a, p)
V_result ← ∅
F = {f ∈ A^F | state(f) == completed ∧
    model(f) is reachable from a}
while (|F| > 0) do
  if |F| > 1 then
    ∃m ∈ F: ∀n ∈ F, n ≠ m:
      time(m) > time(n) ⇒ execute
      compensating activity c(model(m))
  else
    ∃m ∈ F ⇒ execute compensating
      activity c(model(m))
  end if
  F ← F \ {m}
  for each (v ∈ o(c(model(m)))) do
    if (∃w ∈ V_result: model(w) = v) then
      V_result ← V_result \ {w}
    end if
    V_result ← V_result ∪ {(v, c, t)}, c is the
      new value of variable v, t is the
      timestamp of the assignment
  end for
end while
return V_result
```

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

**Definition "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 input data for the re-execution is taken from a variable snapshot that belongs to the start activity or a predecessor of the start activity. The "re-execute" uses the "compensate" operation to reset already completed work in the iteration body.

$$\rho\text{: A} \times P_{\text{all}} \mapsto P_{\text{all}} \qquad (20)$$

The start activity $a \in A$, the process instances $p\_in_g, p\_out_g \in P_{\text{all}}$, and the pre-condition are similar to the iterate operation. The difference is the calculation of $\text{V}^{\text{I}}_{\text{out}}$:

$\rho(a, p\_in_g) = p\_out_g :\Leftrightarrow$

1. $\text{V}^{\text{I}}_{\text{out}} = \text{V}^{\text{I}}_{\text{in}} \cup$ `compensateIterationBody`(a, $p\_in_g$) $\cup$ `loadSnapshot`(b, $p\_in_g$, t, V').

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

An example for the re-execution of activities is given in Figure 5. In a sequence of activities, activity e is currently being executed. The user decides to re-execute the workflow from activity b (Figure 5a). The path from b to the wavefront (activity e) is followed. All links on this path are reset (links b-c, c-d, and d-e) and all activities in the wavefront reachable from b are terminated (activity e). Now, all completed activities in the iteration body are compensated in reverse execution order (Figure 5b). Note that only completed activities with an attached compensation activity can be compensated. Finally, a data snapshot is loaded and the start activity is scheduled (Figure 5c).

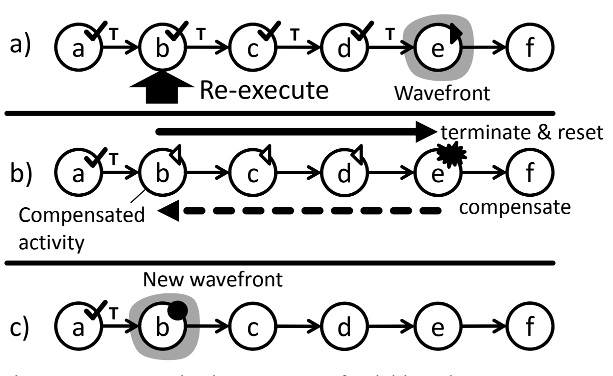


Figure 5. Re-execution in a sequence of activities. The user wants to re-execute activities (a). The iteration body is reset, active activities are terminated, and completed activities are compensated in reverse execution order (b). Finally, a data snapshot is loaded and the start activity for the re-execution is scheduled (c).

In practice, compensation of already completed work is not always possible. An invoked service must provide an operation to undo the results of a former request. For instance, a service with an operation to book a hotel room should also provide an operation to cancel the booking. The ρ operation relies on such compensation operations of services to conduct the compensation of already completed workflow logic in the iteration body. It is up to the person that models the considered workflow to integrate compensation logic in form of a compensating activity c(a) for an activity a. This is a prerequisite for the correct and desired functionality of the re-execution.

## IV. DATA HANDLING

For the repetition of workflow parts the handling of data is of utmost importance. Some of the questions that arise are: 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 appropriate/correct variable values for iterations. Variables might also be reset by the compensation of the iteration body as is done in the "re-execute" operation. This strongly depends on the compensation logic and invoked services. But it cannot be guaranteed that the former variable values are restored by the compensation. Hence, another mechanism is needed.

The desired functionality can be realized by saving snapshots of variables during workflow execution. Available workflow engines store an audit trail [5, 18] that contains, amongst others, values of variables changed by the successfully executed activities. However, the audit trail saves variables incoherently. The proposed snapshot mechanism contains only variables that are visible for a particular activity, their current values and a timestamp. The data footprint of snapshots can be minimized as follows: values for variables that did not change between two snapshots do not have to be stored again; pointers to the values of variables can be used to still be able to refer to them.

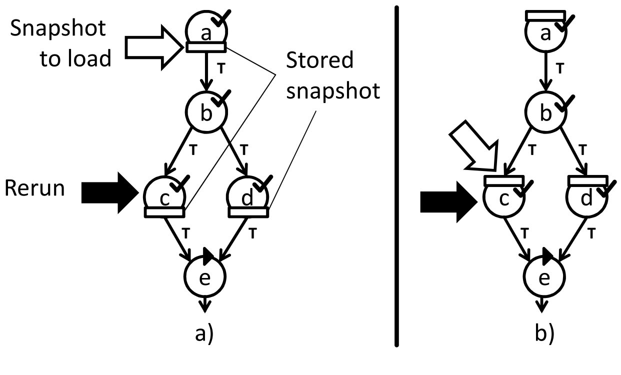


Figure 6. Storing snapshots *after* variable-modifying activities (a) vs. storing snapshots *before* variable-modifying activities (b).

Snapshots are stored with every activity that changes variables. If snapshots are saved *after* variable-changing activities, the workflow graph must always be traversed to find the correct snapshot for an "iterate" or "re-execute" operation. In Figure 6a, the workflow ought to be rerun from activity c. But the nearest previous snapshot of c belongs to activity a. Another approach is to store the snapshots *before* variable-changing activities. In Figure 6b, the rerun starts from the variable-modifying activity c and the snapshot of c can be loaded without a need to traverse the workflow graph. Storing snapshots before the execution of variable-changing activities renders the finding of a snapshot for the start activity of a rerun more efficient.

However, if the start activity of a rerun is a non-variable-changing activity, an algorithm is needed to find the nearest preceding snapshot. The simplest case is a sequence of activities. The snapshot to be considered belongs to the nearest preceding variable-modifying activity. In Figure 7a, the rerun is started from activity b, the corresponding snapshot to load belongs to b's predecessor, activity a. A more complex case arises when there are competing snapshots in a parallel branching. In Figure 7b, the rerun is started from activity e. There are two nearest preceding snapshots located in the branching just before e, one for activity c and one for activity d. This conflict can be solved by the user by manually selecting the snapshot to load. An automatic solution would be to take the snapshot with the youngest timestamp.

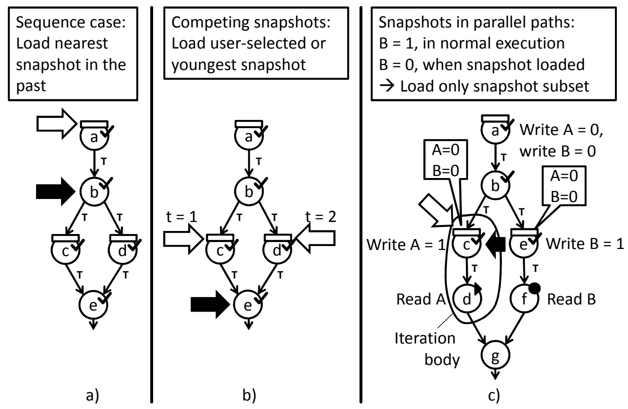

Sequence case: Load nearest snapshot in the past
Competing snapshots: Load user-selected or youngest snapshot
Snapshots in parallel paths: B = 1, in normal execution B = 0, when snapshot loaded → Load only snapshot subset
Write A = 0, write B = 0
A=0 B=0
A=0 B=0
t = 1
t = 2
Write A = 1
Write B = 1
Read A
Read B
Iteration body
a)
b)
c)

Figure 7. Iteration with non-snapshot activities as start activity with unique (a) and competing (b) snapshots. Iterations in parallel paths can cause the problem of lost updates (c).

In parallel paths, the problem of lost updates might occur: loading a snapshot in one path might overwrite the result of a write operation in a parallel path. A simple example is given in Figure 7c. The two global variables A and B are both initialized with 0 by activity a. In the c/d-branch, activity c increases A by 1 and activity d reads A; in the e/f-branch activity e increases B by 1 and activity f reads B. The snapshots of c and e have stored the initial values of A and B. Imagine that c and e are already executed and hence variables A and B have the value 1. Now, the user wants to rerun branch c/d starting with c. The snapshot of c is loaded. Both variables get the value 0, which is correct for A but means a lost update for B. The problem cannot be solved by loading another snapshot in the near environment of activity c (the snapshots of a, c and e have the same content). It must therefore be possible to load only a subset of variables stored in a snapshot. If this is done manually, the user must be able to gain insight into the content of snapshots and to determine the variables to load. An automatic solution is also feasible: all variables that are written in the iteration body can be selected out of the snapshot. In the example, the iteration body consists of activities c and d. The only write operation in the iteration body targets variable A. Hence, variable A can automatically be selected from the snapshot stored before c.

Due to the rerun of activities (manually or in loops) there can be several snapshots for each variable-modifying activity—one snapshot for each execution of the activity. These multiple snapshots are called *snapshot instances*. The user must be given the means to select the particular snapshot instance to be used for the rerun; recommendations may facilitate correctness. In Figure 8, activities c and d of the sample workflow in Figure 7c are iterated multiple times. This leads to a chain of executions of activities c and d. The current value of variable A was taken for each rerun. There are now three snapshot instances for activity c with different values for variable A. Imagine the user wants to iterate again from c. Besides the start activity, he has to select the variables that should be loaded out of the snapshot (variable A) and the concrete snapshot instance. The snapshot instance is identifiable via the execution number of the corresponding activity. For example, the snapshot with A = 100 belongs to the 1st execution of activity c; A = 101 belongs to the 2nd execution of c; and so on. The selected variables of a snapshot instance are re-initialized according to the values stored in the snapshot.

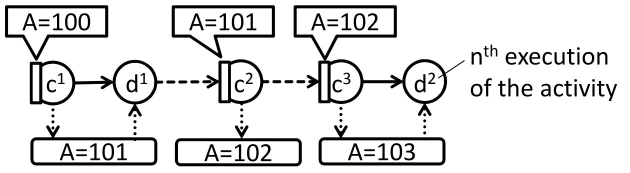


Figure 8. Multiple snapshot instances can exist for one activity.

Now, a function to load snapshot instances can be defined. This function is used by the "iterate" and "re-execute" operations to deliver the correct input for the next enforced run of the corresponding workflow logic.

**Definition "Load Snapshot".** The "load snapshot" function loads variable instances for a process instance and a subset of variables. The signature of the function is defined as follows:

$$\text{loadSnapshot: } A \times P_{\text{all}} \times \mathbb{N} \times \wp(V) \mapsto \wp(V^I). \quad (21)$$

Let $a \in A$ be the activity the snapshot belongs to, $p_g \in P_{\text{all}}$ an instance of process model g, $e \in \mathbb{N}$ the execution number for activity a that identifies the snapshot instance, and $V' \in \wp(V)$ the selected variables to load from the snapshot. Then `loadSnapshot`$(a, p_g, e, V') = \{v_1, \dots, v_n\}$, $v_1, \dots, v_n \in V^I \Leftrightarrow \forall k \in \{1, \dots, n\}$: `model`$(v_k) \in V'$, i.e., variable instances are loaded only for the given variables.

## V. Repetition in Complex Workflow Graphs

In activity sequences, the wavefront consists only of a single element and there are no concurrent and hence no join nodes, which does not pose any complications for iteration and re-execution. This section shows the application of the two ad hoc rerun operations in complex workflow graphs with parallel and alternative branches. The most important issue to solve is to guarantee a correct behavior at join nodes when iterating or re-executing them. In common workflow languages, a join node is activated not until all incoming links are evaluated and the join condition is evaluated to `true`. Consider the example in Figure 9. The join node d has three predecessors, a, b and c. Only if the links a-d, b-d and c-d are followed, and the join condition of d is `true`, can d be scheduled (Figure 9a-c). This join behavior prevents (1) missing link values in the join condition of join nodes, and (2) race conditions, i.e., undesired multiple executions of join nodes or ambiguous behavior.

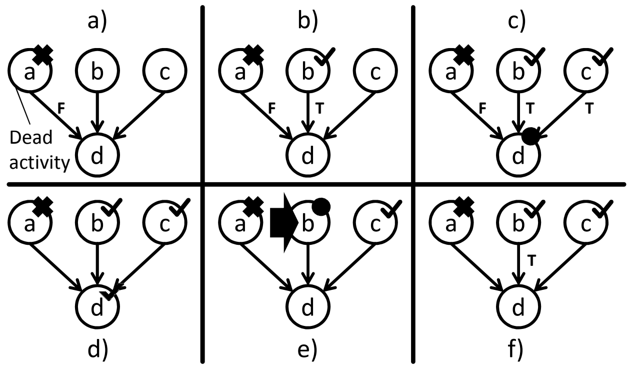


Figure 9. Join behavior of the meta-model. A join activity cannot be executed unless all incoming links are evaluated (a-c). Typically, link states are not stored by the engine (d), which makes a rerun in parallel or alternative branches impossible (e and f).

After the evaluation of the join condition, the values of incoming links of the corresponding activity are not needed anymore. Thus, link values are usually not stored beyond the context of the join node (Figure 9d). This is the typical way of dealing with links in conventional workflow engines. In Petri net-based workflows [19] and BPMN [14], link values are tokens that get consumed by the transition of a join node or a gateway, respectively. In BPEL [15], the link values are bound to boolean variables that are visible only in the context of the target activity instance. That means these variables are destroyed after the execution of the target activity and hence the old link states are lost. As a consequence, a join activity in the iteration body of an "iterate" or "re-execute" operation can lead to a deadlock. In Figure 9e, the iteration body consists of activity b and the join activity d. After the execution of b activity d would never be executed because of the missing states of links a-d and c-d (Figure 9f). Storing the set of evaluated links $L^E$ in the context of process instances (see Definitions "Evaluated Links" and "Process Instance") helps solving this problem. The following different use cases show the application of the concept in different situations.

## *A. Start activity is in a completed AND-branching*

The first case discussed is the one in which the start activity for the rerun is located in a parallel, already completed branch. That means the join activity that closes the branching is at least scheduled. Figure 10 shows an example of this case. The parallel branching of activities b to g is completed. The user requests an iteration or re-execution from activity c (Figure 10a). The path beginning with c is followed forward to the wavefront (Figure 10b). All links on this path are reset; all activities in the wavefront reachable from c are terminated (activity h). In case the user wants to re-execute the workflow logic, all completed activities on the path from the start activity c to the wavefront that have assigned a compensation activity (here: c, e and g) are compensated in reverse execution order. Note that the other path of the considered parallel branch, containing activities d and f, and the branch i to m remain unchanged. Finally, start activity c is scheduled (Figure 10c). The wavefront now consists of the activities l and c and the link f-g. In the course of the further execution of the process instance the join activity g can run normally since the state of link f-g is stored in $L^E$.

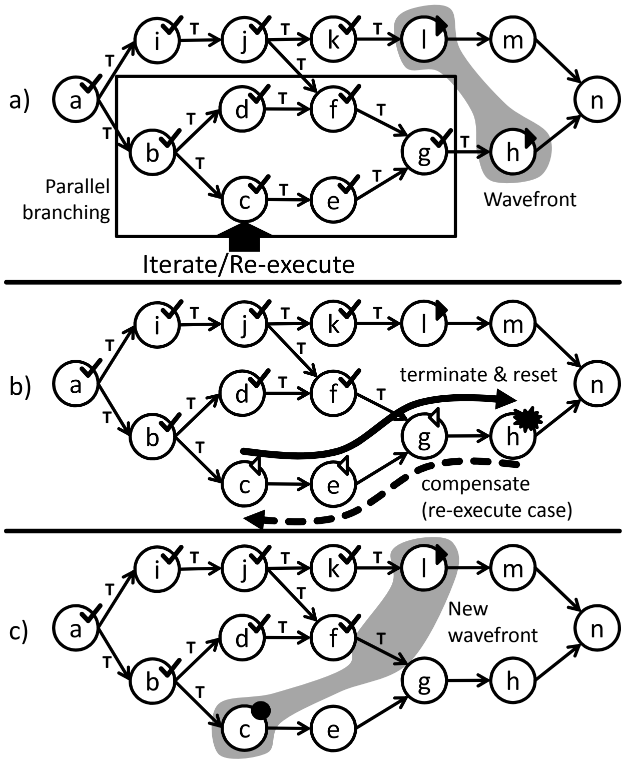


Figure 10. Rerun in an AND-branching.

## *B. Start activity is in a completed XOR-branching*

The rerun in an already completed XOR-branching is very similar to the AND-branching case. In the meta-model, an XOR-branching is achieved with the help of mutual excluding transition conditions of outgoing links of split nodes. In Figure 11a, the link b-c was evaluated to `true`, whereas b-d is `false`. Hence, activities b through g implement an alternative branch. The path containing the link b-d, the activities d and f and the link f-g is dead. The join behavior in the meta-model requires all links to be evaluated before a join node can be executed. That is the reason why an algorithm for dead path elimination (DPE) is used to set all links in a dead path to `false` [5]. In the example, this holds for the links d-f and f-g. Activities on a dead path are not executed; their state is simply set to `dead` (activities d and f). In this scenario, the user wants to rerun the workflow from activity c which is located in the completed path of the XOR-branching. Due to the DPE and the set of all evaluated links $L^E$, this case can now be addressed exactly the same way as the ad hoc rerun in completed AND-branches.

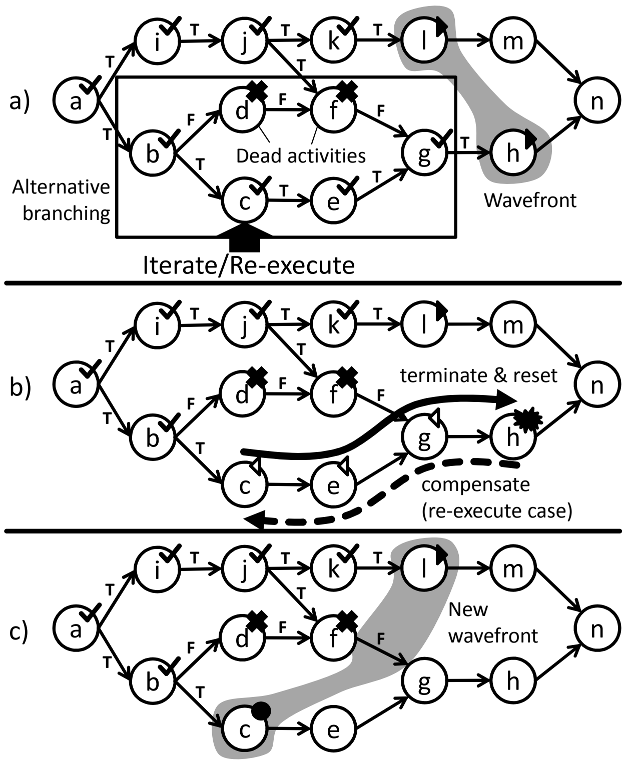


Figure 11. Rerun in an XOR-branching.

Note that there are cases where a branching in a process model can be an AND in some process instances and an XOR in other instances. It depends on the selected transition conditions and the process context (e.g. variable values) if one, all or a particular number of branches is followed during workflow execution. In BPMN, this behavior can be modeled with an inclusive gateway [14]. However, such cases are covered by the concept because links in dead branches are evaluated (to `false`) in the course of the DPE.

## C. Start activity is before a branching

In this case, the start activity is located before a branching, i.e., the iteration body contains branching activities. In Figure 12a, the user reruns the workflow with b as start activity. The two outgoing links of b show that it is a branching activity. In order to address this case, all paths beginning with b are followed to the wavefront (Figure 12b). All visited links are deleted from $L^E$ (b-d, d-f, f-g, b-c and c-e) and all visited scheduled or running activities are terminated (e). In the re-execute case, the reachable completed activities that can be compensated (c and d) are compensated in reverse execution order. After that, the start activity is scheduled (Figure 12c) and the workflow can be resumed.

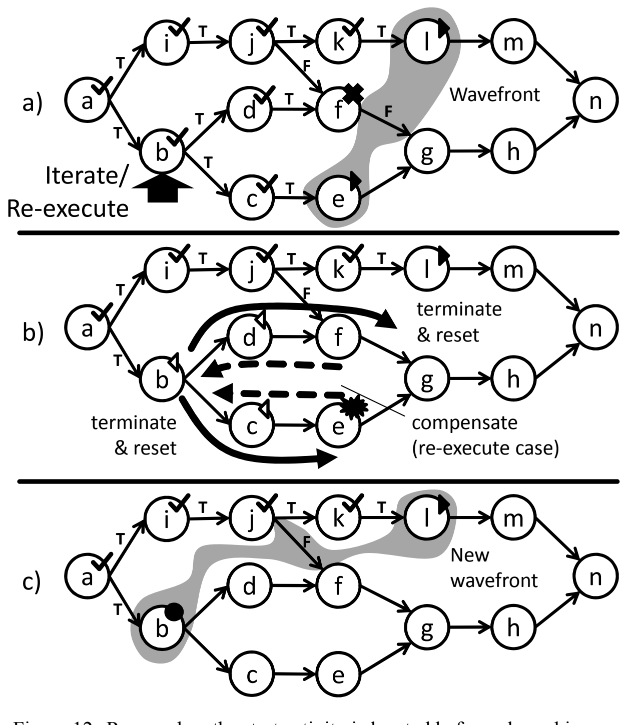


Figure 12. Rerun when the start activity is located before a branching.

Figure 13a shows a more complex scenario. Note that the difference to the previous workflows is that link j-f was deleted and link d-k added. The user wants to rerun the workflow beginning with activity b. The iteration body thus contains two branching (b and d) and two join activities (g and k). All links on the path from b to the wavefront are reset and all scheduled/running activities are terminated (Figure 13b). It is sufficient to visit the activities and links only once with the algorithm, like activities g and h and link g-h. In the re-execution case, the completed activities on the considered path are compensated in reverse execution order (g, f, e, d, and c). The start activity (b) is then scheduled and the rerun operation is complete (Figure 13c).

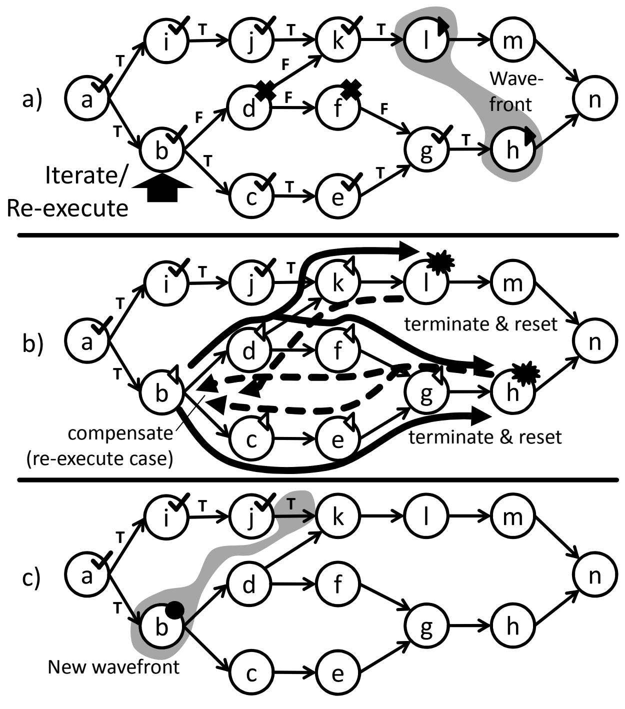


Figure 13. Complex rerun scenario with several branches.

## D. Repetition in Dead Paths

As shown above the concept to enforce the rerun of workflow logic can be applied in sequences of activities, in parallel and alternative branches, and in complex graphs. An interesting research question is (1) whether the start activity of a rerun can be located in a dead path and (2) whether such an operation would be meaningful. Note that there is a difference between a dead path and a path that is not (yet) executed. A dead path belongs to the past of the workflow instance while a not executed path is the future of the workflow instance. That means the latter is a jump to the future, that can be realized by a “skip” operation, which, however, is not part of this work.

The precondition of the “iterate” and “re-execute” operation is that the state of the start activity is scheduled, executing, completed, faulted or terminated (see Section III.A). The iteration/re-execution in dead paths is thus not allowed. However, if this precondition was neglected, repeating activities in a dead path would technically be possible with the presented concept. As an example consider Figure 14a. The user requests the repetition of activity f, which is located in a dead path. As usual, the path from f to the wavefront is followed, links are reset (f-g and g-h), running activities terminated (h) and completed activities compensated (g, in case of a re-execute), as shown in Figure 14b. Then, the start activity f is scheduled and can be executed when the workflow is resumed. Although conceptually feasible, the operation has several problems. The result is obviously an unrealistic execution history. Activity f gets executed although its predecessors d, i and j were not enacted (Figure 14c). Further, the operation is not really an iteration or a re-execution because at least the start activity was not executed before the operation. Hence, it is not a repeated execution of activities but rather an ad hoc change of a process instance that enforces the execution of a dead path. That is why the above-mentioned precondition prevents a repetition of activities in dead paths.

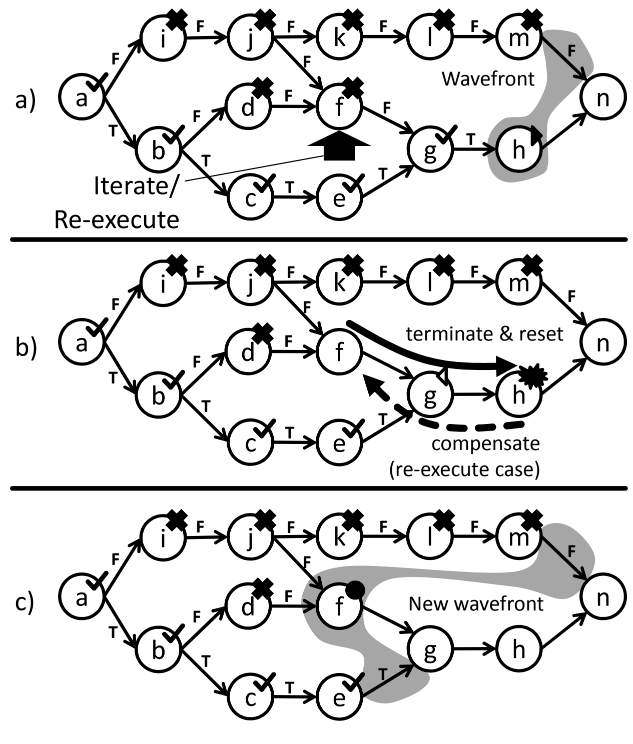


Figure 14. Rerun in a dead path.

However, although it is not recommend using the "iterate/re-execute" operation in dead paths, a workflow system implementing the approach should provide as much flexibility as possible for scientists or other users. The user must decide whether the operation helps to achieve the desired goals. With the help of the precondition the workflow system is able to detect that the user is about to conduct an ad hoc rerun in a dead path. The user should then be requested if he really wants to apply the operation in a dead path and if so, the system conducts the operation as shown in this section.

## VI. Implications on the Execution Correctness

Workflows consist of different activity types, e.g., for sending/receiving messages, loops, or variable assignment. The enforced repetition of workflow logic has to account for different activity types, especially those that interact with external entities such as clients, humans, or services/programs. The main problem is that the repetitions are not reflected in the workflow logic because they are an ad hoc user operation. Hence, the aforementioned external entities do not know a priori the exact behavior of the workflow. An uncoordinated rerun of workflow logic can lead to multiple invocations of services, multiple identical work items in the work list for human users or an infinite waiting for messages because the communication partner does not know that a message must be sent again.

### *A. Message-receiving Activities*

If a message receiving activity is repeated, it would wait infinitely for the message because it was already consumed. Three cases can be distinguished to solve this problem. Firstly, the original message sent by the partner in a former iteration is taken as incoming message for the next run of the activity. However, if the activity was iterated several times, there may be different versions of the incoming message. The user then has to select the desired message.

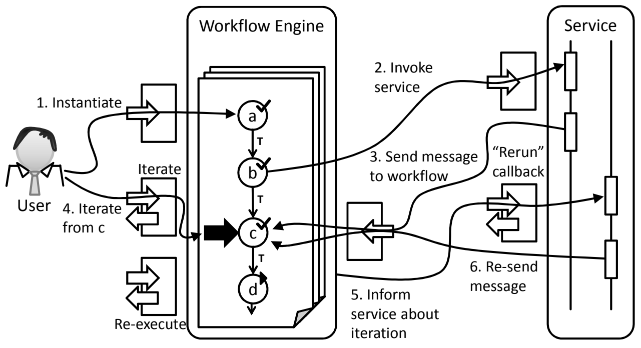


Figure 15. Repetition of a message-receiving activity. The communication partner is informed about the rerun of the activity over a special "rerun callback" operation. The partner then can re-send the message or send an adapted (i.e., updated) message to the engine.

Secondly, the message sending partner re-sends the message or sends an adapted message. The partner needs to be informed about the repetition of the activity. A simple solution is that partners provide a special "rerun callback" that can be used by the workflow engine to propagate the iteration or re-execution. An architecture for this concept is shown in Figure 15. The workflow engine provides the "iterate" and "re-execute" operations. The considered workflow that is deployed on the engine implements two operations, one for its instantiation and one to receive a message from a service. The course of action in this setting is as follows. The user invokes (i.e., instantiates) the workflow (1). The workflow calls a matching service in an asynchronous manner (2) and provides a callback for the response (activity c). The invoked service creates the response message and sends it to the engine (3). The engine correlates the message to the particular workflow instance (e.g., via the instance id or some other information that uniquely identifies the workflow instance). Now the user decides to iterate the workflow logic with activity c as start activity and invokes the corresponding "iterate" operation (4). The workflow then waits again for the message of the service. The engine informs the partner about the iteration that took place in the workflow instance. This is done by invoking the special "rerun callback" provided by the partner (5) or a mechanism in the service infrastructure performing the same functionality. The engine's message contains at least the following information: the original message of the partner (in case the partner did not persistently store the message), the engine's address, and correlation information to identify the workflow instance. The partner then decides whether to re-send the message or to send an adapted one (6). Sending an adapted message is useful if the information distributed by the partner has to be updated (e.g., sensor data). The engine has to find the partners to be considered in order to invoke their "rerun callback". It first searches for all message-receiving activities in the iteration body. Then, it determines the addresses of the related partners. The addresses can be found in the "ReplyTo" header field of a message received in a former run of the workflow logic (if WS-Addressing [20] is used). Or it is taken from a message that was sent to a partner by a message sending activity in the same workflow instance. This scenario has the disadvantage that it has many implications on the partners' services and/or infrastructure and would be difficult to enable in a standard manner.

Thirdly, message-receiving activities are usually related to message-sending activities. The workflow system or the user pays attention that if a message-receiving activity is iterated, its corresponding message-sending activity is iterated, too. The reason is that an incoming message is often the response to a message sent to a partner. Hence, repeating the invocation of the partner will make the partner send the message again to the workflow engine (or an adapted message with updated content).

A workflow system that implements the ad hoc rerun should support all three cases. It depends on the implemented message exchange pattern, on the concrete function realized by the partner and on whether the partner is stateful or stateless to select (possibly with user-support ) the adequate mechanism for the repetition of message-receiving activities.

### B. Message-sending Activities

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.

### C. Loops

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 (e.g., in a process repair component). This functionality can be used to resolve infinite loops.

It can also happen that the start activity of an iteration or re-execution is located within an already completed loop. The operation causes the loop to run again. In its first iteration the loop body begins with the start activity of the ad hoc rerun. From its possible second iteration on the complete loop body is executed. The user must be able to select the particular iteration of the loop that should be repeated. This can be done with the help of a variable snapshot loaded prior to the rerun because the former variable values represent the iteration of a loop (e.g., via a counter variable).

### D. Workflow Languages with Block Structures

In practice, a workflow is not a simple graph consisting of nodes and edges. There are often also hierarchically nested elements. In BPMN, there is the concept of sub-processes that are containers for arbitrary workflow logic [14]. BPEL offers structured activities (e.g., the `sequence`, `flow`, or `while` activity) that can contain other activities for a simplified modeling of complex workflows [15]. The rerun of activities in hierarchical structures has to account for parent-child-relationships of activities.

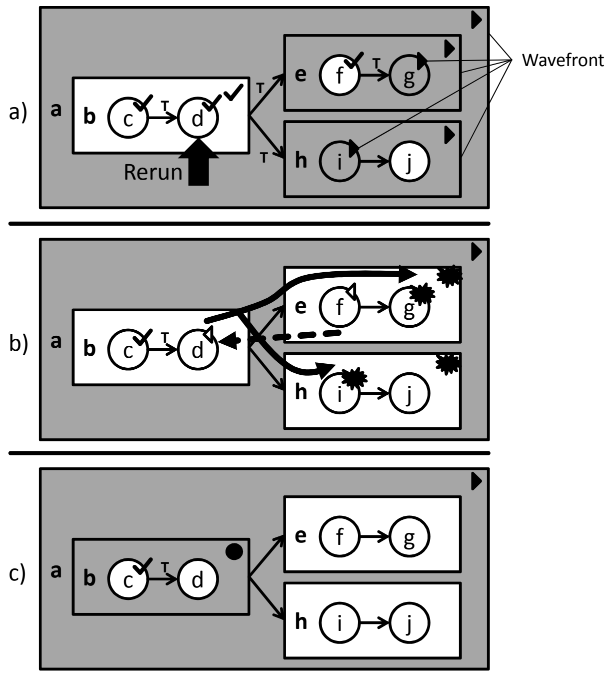


Figure 16. Rerun in workflows with block-structures

Figure 16 illustrates an example for the rerun in hierarchically nested activities. The process model contains a sequence of activities c and d followed by a parallel branching of f/g and i/j (Figure16a). Because of the nesting, the wavefront of the considered process instance (the shaded activities) is stretched across the complete process. All parent activities with at least one active child are also active. Hence, the termination and resetting of the path from the start activity to the wavefront is more complex. In the example, d is the start activity of the rerun (Figure 16a). The atomic activities d, f, g and i as well as the parent activities b, e and h have to be terminated or reset (Figure 16b). Moreover, the start activity cannot be simply scheduled. It must be checked whether the corresponding parent activity is active. If so, the start activity can be scheduled. If not, the parent must be scheduled instead of the start activity itself (and possibly the parent of the parent, etc.). It is important to make sure that, although the parent activity is scheduled, only a subset of its child activities will be executed. In Figure

16c, the parent activity b of the start activity d is scheduled. But it must be ensured that activity c is not executed again. Realizing this behavior in a workflow engine is highly implementation-specific.

### *E. Impact on Scopes*

In modern workflow languages such as BPMN or BPEL, the concept of scopes is used to denote containers for activities, data objects and correlation keys; they span transaction boundaries and specify fault handling logic as well as logic to handle incoming events. At the beginning of their execution, scopes initialize their context. That means fault handlers and event handlers are installed and local variables are instantiated.

If a rerun is conducted with the start activity being located in an already completed scope, this scope has to be scheduled because of the parent-child relationship discussed before. The scope's context has to be initialized again. In case of a re-execution, the scope's effects have to be undone before the workflow can be resumed. Invoking the scope's compensation handler undoes the work of the complete scope. This is the desired behavior only when the start activity is the first activity in the scope. Otherwise the specific compensation handler of the scope must not be executed, but rather only the compensation handlers of the activities following the identified start activity in the reverse execution order.

The repetition of activities also has an impact on fault and compensation handlers attached to scopes. Fault and compensation handlers can be used to undo already completed work. If logic is rerun within these handlers, it must be ensured that the corresponding scopes are not compensated multiple times.

## VII. User Interaction with the Workflow System

A workflow system that implements the ad hoc rerun of workflow logic must provide a monitoring tool that allows users to continuously follow the execution state of process instances (see Figure 17(1)). The user interacts with the system as follows. If the user detects a faulty or unintended situation, he can suspend the workflow (2) and manually trigger an iteration/re-execution (3).

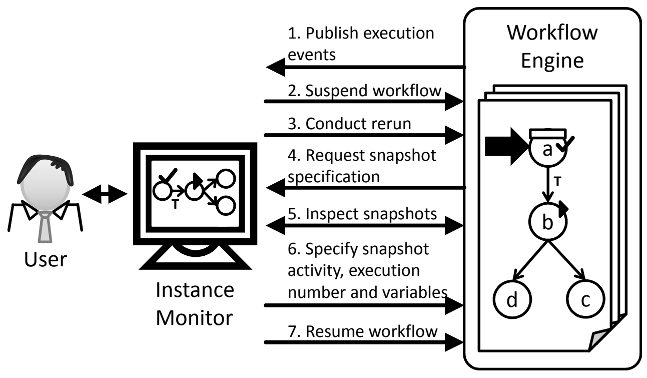

User
Instance Monitor
Workflow Engine
1. Publish execution events
2. Suspend workflow
3. Conduct rerun
4. Request snapshot specification
5. Inspect snapshots
6. Specify snapshot activity, execution number and variables
7. Resume workflow
a
b
c
d
T

Figure 17. User interaction with a workflow system that implements the concept of the enforced repetition of workflow logic

The workflow system asks him to specify which snapshot instance should be taken for the rerun and which contained variables should be loaded (4). The user can inspect different snapshot instances and the values of their variables in order to determine the desired snapshot instance (5). He specifies the snapshot instance with the corresponding variable-modifying activity, the execution number of the activity, and the subset of variables to be loaded. The process instance state is changed in the engine as described in Section III through V. Finally, the user resumes the workflow instance (7). Note that steps 4 to 6 are omitted if the user conducts an iteration of activities with the current variable values, i.e., without loading a snapshot.

## VIII. Implementation

The implementation of the "iterate" and "re-execute" operations is based on the Apache Orchestration Director Engine (ODE) [21] as BPEL engine and on the Eclipse BPEL Designer [22] as GUI for the users of the system.

### *A. Architecture of the System*

Figure 18 shows the high level architecture of the workflow system that implements the ad hoc rerun of workflows. Components with dashed lines are new or extended. The scientist/user interacts with Eclipse and the BPEL Designer plugin in order to model and run workflows. The *Execution Control* component enables starting of workflows directly in the BPEL Designer. A special dialog requests the user to specify the content for the input message. Deployment of workflows happens transparent for the user. The underlying workflow engine is hidden. Workflow instances can be suspended and resumed. The *Instance Monitor* visualizes the current execution state of running workflows by coloring activities and links. The scientist can inspect and change values of variables and endpoint references assigned to partner links.

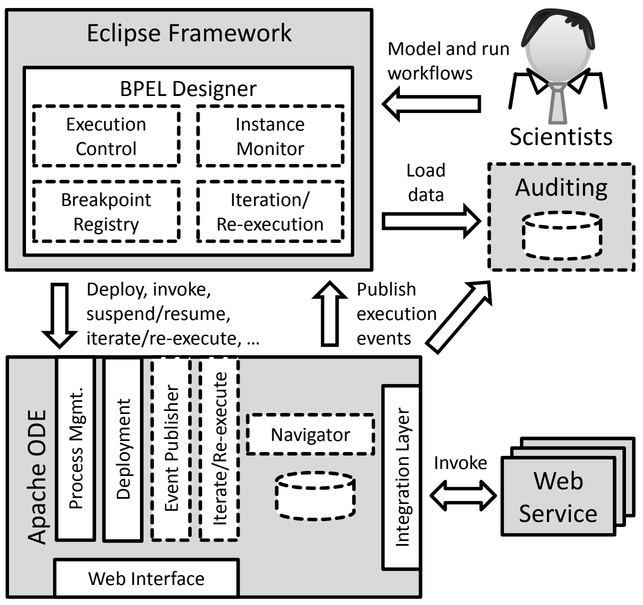


Figure 18. High level architecture of the prototype

In order to suspend workflows at points of particular interest for the user, it is possible to set breakpoints at activities or links in a *Breakpoint Registry*. The *Iteration/Re-execution* component provides the ad hoc rerun operations to the user. A wizard helps the user to find and select the desired snapshot instance to load prior to the rerun. The needed information is fetched from the workflow engine. When a rerun is conducted, the activity states of the instance monitor are refreshed and an iterate/re-execute operation of the engine is invoked.

Apache ODE provides interfaces to deploy and undeploy processes (*Deployment* component) and to access information about process models and instances (*Process Management* component). Web services are invoked over an *Integration Layer*. There is also a *Web Interface* for user, which does not play a role in this work. The Apache ODE was extended with an *Event Publisher* that emits execution events of workflow instances, e.g., activity ready, activity running, or link evaluated. These events are received by the BPEL Designer's Instance Monitor and used to color activities and links. The *Navigator* is the heart of the workflow engine. It traverses the workflow graph and executes activities. An extension of the Navigator and the database is that variable and partner link values are stored as snapshot before the execution of variable changing activities. The new *Iterate/Re-execute* component provides the two rerun operations to clients. The component loads a variable/partner link snapshot according to the input of the user. Then, the execution queue of the navigator is adapted: activity instances that have to be terminated are removed from the execution queue; a new instance of the start activity is scheduled (and possibly new instances of its parent activities), i.e. put to the execution queue.

An *Auditing* component external to the workflow engine stores the published execution events persistently. The BPEL Designer makes use of the Auditing to load the state of a workflow instance into the Instance Monitor. This has the advantage that the engine's execution events are not lost even if Eclipse is shut down during workflow execution.

The following two sections provide more details on the extensions of the BPEL Designer and the Apache ODE.

### B. *Extensions of the BPEL Designer*

The scientist can use the functions of the Execution Control from the extended toolbar menu (Figure 19a). A workflow can be started, suspended, resumed and terminated. If a breakpoint is reached during execution, a skip breakpoint operation releases the breakpoint and the workflow execution proceeds.

In order to implement the Instance Monitor the Eclipse Modeling Framework (EMF) ecore model for BPEL was extended with a `state` attribute for all activities and links. It holds the state of activities/links based on the execution events of the engine. The state indicates the color of each element (Figure 19b): yellow is running, green is completed, red is faulted, orange means a breakpoint is reached, and grey are dead activities.

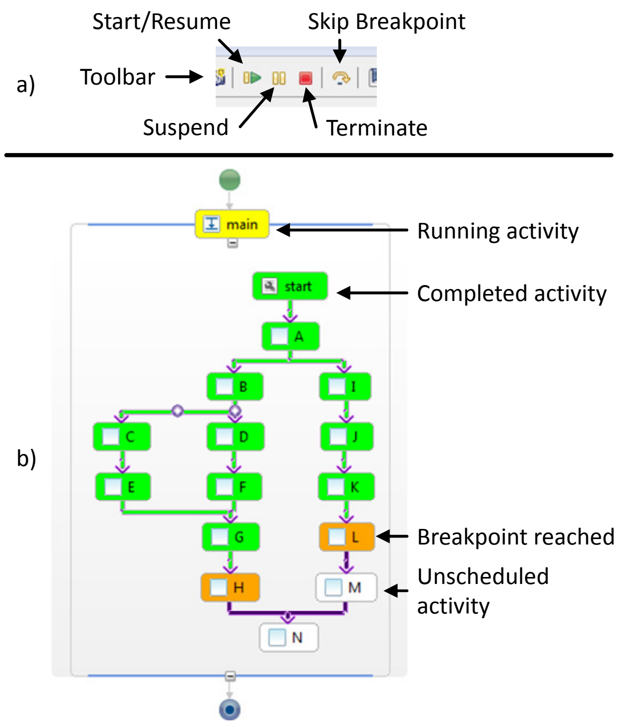


Figure 19. BPEL Designer extension: (a) Execution Control in the toolbar and (b) the Instance Monitor.

When a workflow is suspended, the user can iterate or re-execute workflow logic via the context menu of an activity (Figure 20). The selected activity is then the start activity of the operation (activity B in the figure).

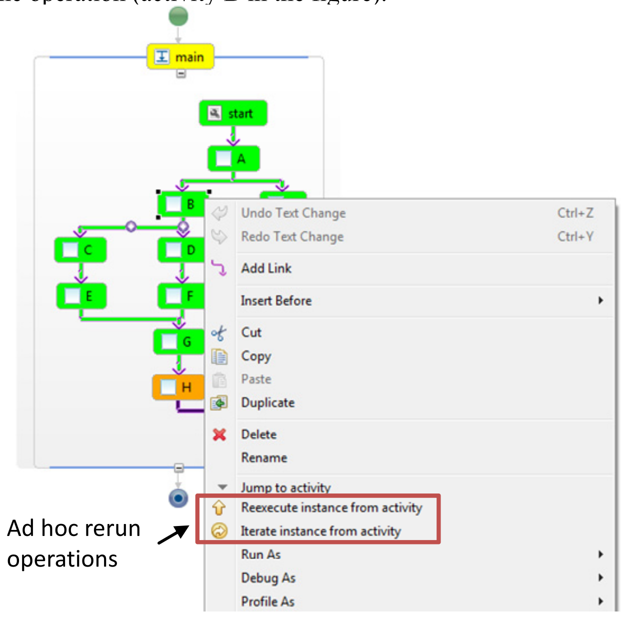
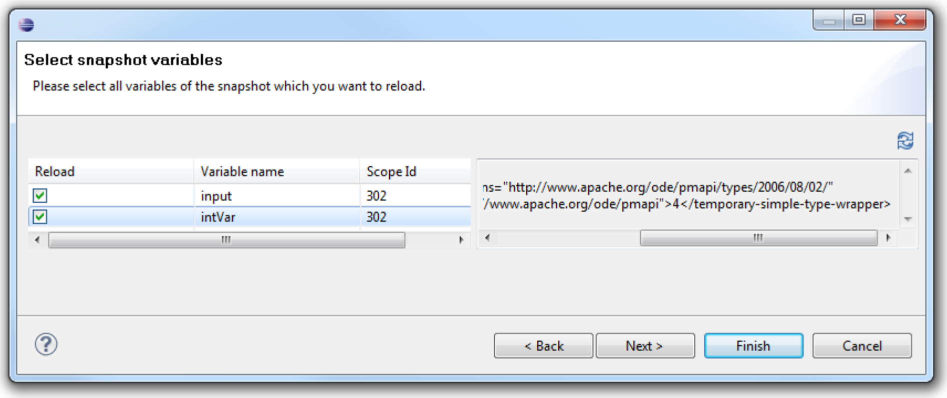


Figure 20. The user can select the iterate/re-execute operations from the context menu of activities.

A wizard opens that guides the user step by step through the snapshot selection process. First, the activity to load the snapshot for has to be chosen. This can be the start activity or a predecessor thereof. The latter can happen when the start activity is no variable-changing activity and hence does not possess a snapshot. The wizard shows all snapshot instances for the selected activity.


Select snapshot variables

Please select all variables of the snapshot which you want to reload.

| Reload | Variable name | Scope Id |
|---|---|---|
| ✓ | input | 302 |
| ✓ | intVar | 302 |

```
ns="http://www.apache.org/ode/pmapi/types/2006/08/02/"
'/www.apache.org/ode/pmapi">4</temporary-simple-type-wrapper>
```

Figure 21. Wizard to select variable snapshots.

The user can have a look at the snapshot content, i.e., at the values of stored variables (Figure 21) and partner links. It is possible to select only a subset of stored variables (Figure 21) and partner links to be loaded in the course of the rerun, which can prevent a lost update of variables in parallel paths. When the snapshot selection is done, the Instance Monitor is refreshed, i.e., the state of all activities in the iteration body is set to inactive. Finally, the iterate/re-execute operation of the Apache ODE is invoked.

## C. *Extensions of the Apache ODE*

The Navigator was extended so that each variable-changing activity persistently stores a snapshot with all visible variables and partner links before its execution. This pertains to `receive`, `pick`, `invoke` and `assign` activities. Three new tables are created in the database to store the snapshots (Figure 22). The table ODE_SNAPSHOT holds information about snapshot instances: the corresponding process instance, the scope the stored variables and partner links belong to, the creation time, the version, and an XPath expression pointing to the corresponding activity. The table ODE_SNAPSHOT_VARIABLE stores the concrete values of variables that belong to a snapshot. And finally, the table ODE_SNAPSHOT_PARTNERLINKS holds the values of partner links stored in snapshots. A partner link can have up to two values, one EPR for each of the at most two roles. There is another new table, ODE_LINK_INSTANCE, used to save the state of link instances as discussed in Section V.

The Web service interface of Apache ODE was extended with five operations. The `iterate/re-execute` start the ad hoc rerun for a specific workflow instance. Both require the process instance ID, the XPath expression of the start activity, the XPath expression of the activity to load the snapshot for, the snapshot version (i.e., the execution number), and a list of variables and partner links to load. The `getSnapshots` operation delivers all snapshot instances for a given process instance and activity, but without loading the concrete values of the stored activities/partner links; the `getSnapshotPartnerLinks` and `getSnapshot-Variables` operations are then used to load concrete values out of a snapshot identified via the process instance and snapshot ID. This functionality is distributed on several operations for the sake of smaller messages. It is often sufficient to load just some general information about a snapshot and not all the contained values.

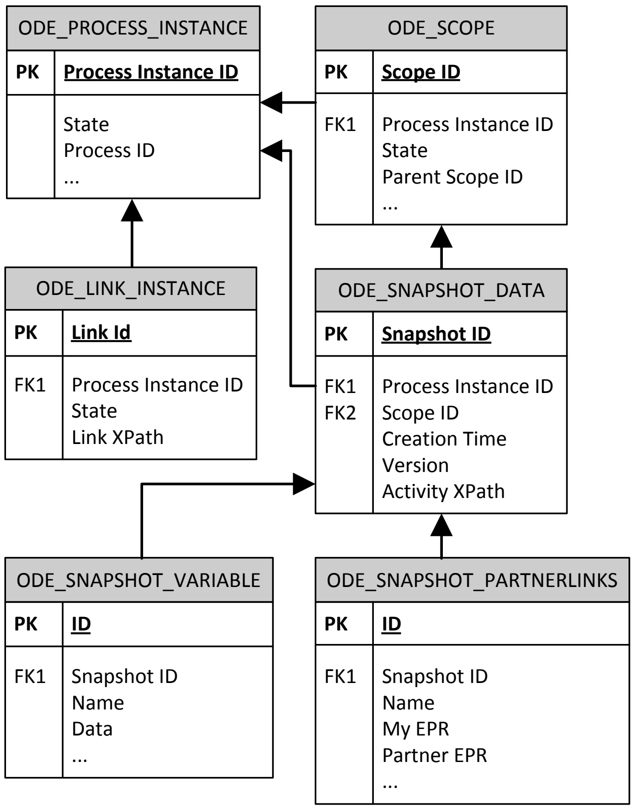


Figure 22. Extension of the database schema to store variable/partner link snapshots and the state of link instances.

The two most critical parts of the ad hoc rerun are (1) to correctly and consistently adapt the content of the execution queue and (2) to adapt the activity instance of the start activity's parent. In Apache ODE, the execution queue is an object that holds a list with all scheduled activity instances, another list with all channels used to send information between activities (e.g. a child activity uses a channel to inform its parent about its completion), and a third list with completed activities. All activity instances and channels that belong to activities in the iteration body have to be removed from these lists.

The modification of the start activity's parent has to be implemented per activity type. It is currently realized for `sequence` and `flow` activities. In a `sequence`, the `sequence` activity instance is scheduled again but only with the start activity and all successor activities as children. All activities preceding the start activity are omitted because they do not belong to the iteration body. In a `flow`, all completed activities of the former iteration have to be marked as *not completed* and are scheduled again. Their activity guards make sure that the activities are executed not until their join conditions can be evaluated. Only the start activity of the rerun is executed without evaluation of its join condition.

## IX. Related Work

The term "iteration of activities" is mentioned in [5] as one of the change operations that can be performed in a workflow; no details are available about how iteration should be performed. In ADEPT, it is possible to perform manual ad hoc backward jumps that are similar to the rerun operations in this paper, as claimed in [16]. The target activity of the jump is executed again. The previous execution state is restored based on the execution and data element history. While in [16] it is said *that* an operation for ad hoc backward jumps exists, no details such as algorithms, applications on workflow languages with hierarchically nested elements, or impact of different activity types are provided as is done in this work. In the scientific workflow system e-BioFlow, scientists can re-execute manually selected tasks with the help of an ad hoc workflow editor [6]. 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. Following the approach in this paper, 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.

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 [23] for the case of business transactions. Eberle et al. [10] apply this concept to BPEL scopes. In BPMN [14] this behavior can be modeled with sub-processes, error triggers and links. In IBM MQSeries Workflow a Flow Definition Language (FDL) activity is restarted if its exit condition evaluates to `false`. ADOME [24] can rerun special repeatable activities if an error occurs during activity execution; the approach is applicable only for single activities, not for groups of activities. In Apache ODE, an extension of BPEL's `invoke` activity enables retrying a service invocation if a failure happens [25]. These approaches have special modeling constructs in common to realize the repetition. In these cases, the rerun is pre-modeled at design time. In contrast to these approaches, the solution in this paper aims at repeating a workflow starting from an arbitrary, not previously specified point.

Iterations can also be realized by configuring workflow models with deployment information. Invoke activities in the Oracle BPEL Process Manager [11] 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 [26] and also in [27] where the service selected for the retry is identified using a semantic description of selection criteria. The scientific workflow system Taverna [28] allows specifying alternate services that are taken if an activity for a service invocation fails. In contrast to these and other available similar approaches, this paper advocates a solution where the rerun can be started spontaneously without a pre-configuration of workflows from an arbitrary point.

The scientific workflow system Pegasus can automatically re-schedule a part of a workflow if an error occurs [12]. Successfully completed tasks are not retried. The Askalon workflow system provides a checkpointing-like functionality to handle runtime faults [29]. Kepler's Smart Rerun Manager can be used to re-execute complete workflows [30]. Tasks that produce data that already exists are omitted. The main difference of these approaches to this paper is that the ad hoc rerun allows selecting the starting point of the iteration (manually) and hence this functionality can be used for different purposes, e.g. explorative workflow development, steering of the convergence of scientific results, or fault handling.

Checkpointing in workflow management is a technique to store the complete workflow state at specific execution points geared towards transactions spheres [31]. If a failure happens, these checkpoints can be used to rollback a workflow, i.e., load its former state, and run a part of the workflow again. Assurance points (AP) [32] are a similar concept that store data at critical points in a workflow. APs are user defined at modeling time and enable backward recovery of a complete process, retry of a workflow part, and forward recovery. Compared to the approach in this paper, checkpoints and assurance points cannot be used to rerun a workflow part starting from an *arbitrary* activity chosen at runtime. Apart from this, the retry functionality of APs can be compared to the re-execute operation in this work because already completed work from the current wavefront to the AP is compensated. In [33], an aspect-oriented approach for dynamic checkpointing in workflows is introduced. It allows selecting and changing checkpoint positions at workflow runtime in order to transfer running workflows from one to another workflow engine instance. The approach can be used to rerun activities of a workflow in an ad hoc manner. In contrast to the approach in this paper, the rerun would require an additional step: the selection of an adequate checkpoint in the future of a workflow instance that will be the target of a rerun later on. Thus, the scientist must prepare a rerun before the execution of the workflow part, which is more restrictive than the ad hoc rerun proposed in this work.

In [34], the authors present and describe several types of flexibility in process-aware information systems. The option "Undo task A" in the flexibility type "Flexibility by deviation" is similar to the iterate/re-execute operation in this work. The control is moved back just before the execution of a task (= iteration); in some cases, it is meaningful to compensate already completed work (= re-execution). No further details are provided about data issues, how race conditions are avoided, how parallel/alternative/dead paths are dealt with, or how block-structures influence the approach as it is done in this work.

## X. Conclusion And Future Work

This paper dealt with the formal description of two operations to enforce the rerun of workflow logic during workflow execution: 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. It was shown that the approach can be applied in sequences of activities, parallel and alternative branches as well as in more complex scenarios that include the repetition of join activities. Furthermore, an adoption of the operations in dead paths has been investigated. An ad hoc rerun in dead paths is not recommended because it is literally no rerun of activities. But it should be up to the user to decide about the meaning of such an operation. One of the main issues when repeating activities is the question which data to take as input for the next run. This issue is addressed with the help of data snapshots that are stored before each variable-modifying activity and that are loaded in the course of the rerun.

Real world processes depend on external communication partners, services or clients. An operation for the repetition of activities has to account for dependencies on messages from partners and on the impact of repeatedly delivered messages on services invoked by the workflow. There are three ways to deal with the repetition of message-receiving activities: reuse a message received in a former iteration, inform the communication partner about the ad hoc rerun and the partner re-sends the message, and repeat a message-receiving activity together with its corresponding preceding message-sending activity. Furthermore, it was shown how users interact with such a flexible workflow system. A workflow instance monitor that shows the workflow progress in real-time and that allows an immediate intervention of the user is of utmost importance in this setting. The concepts presented in this paper are based on an abstract meta-model and thus can be applied to existing or future workflow engines and languages. It was shown how the ad hoc rerun works in languages with concepts for block-based modeling and scopes, such as BPEL or BPMN. The implementation of the iterate and re-execute operations for BPEL in the Eclipse BPEL Designer and Apache ODE evaluate the formal concepts presented in this paper and proof their feasibility.

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

In future, we will also work on an ad hoc "skip" operation that allows omitting activities, e.g., if the result of the respective activities is already present.

## 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] M. Sonntag and D. Karastoyanova, "Enforcing the Repeated Execution of Logic in Workflows," Proc. of the 1st International Conference on Business Intelligence and Technology (BUSTECH 2011), 2011.

[2] W. M. P. 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 Conference on Enterprise Information Systems (ICEIS), 1999, pp. 353–360.

[3] M. Reichert and P. Dadam, "ADEPT$_{flex}$—Supporting dynamic changes of workflows without losing control," Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems, vol. 10(2), 1998, pp. 93–129.

[4] F. Casati, S. Ceri, B. Pernici, and G. Pozzi, "Workflow evolution," Journal of Data and Knowledge Engineering, Elsevier, vol. 24(3), 1998, pp. 211–238.

[5] F. Leymann and D. Roller, "Production workflow—Concepts and techniques," Prentice Hall, 2000.

[6] I. Wassink, M. Ooms, and P. van der Vet, "Designing workflows on the fly using e-BioFlow," Proc. of the International Conference on Service Oriented Computing (ICSOC), 2009.

[7] R. Barga and D. B. Gannon, "Scientific vs. business workflows," in: I. Taylor, E. Deelman, D. B. Gannon, and M. Shields (Eds.), "Workflows for e-Science—Scientific workflows for grids," Springer, 2007, pp. 9–18.

[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, 1998, pp. 145–164.

[9] M. Sonntag and D. Karastoyanova, "Next generation interactive scientific experimenting based on the workflow technology," Proc. of the 21st IASTED International Conference on Modelling and Simulation (MS), 2010.

[10] H. Eberle, O. Kopp, F. Leymann, and T. Unger, "Retry scopes to enable robust workflow execution in pervasive environments," Proc. of the 2nd Workshop on Monitoring, Adaptation and Beyond (MONA+), 2009.

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

[12] E. Deelman, G. Mehta, G. Singh, M.-H. Su, and K. Vahi, "Pegasus: Mapping large-scale workflows to distributed ressources," In: I. Taylor, E. Deelman, D. B. Gannon, and M. Shields (Eds.), "Workflows for e-Science—Scientific workflows for grids," Springer, 2007, pp. 376–394.

[13] H. Garcia-Molina and K. Salem, “Sagas,” Proc. of the ACM Sigmod International Conference on Management of Data, pp. 249–259, 1987, doi:10.1145/38713.38742.

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

[15] OASIS, “Web Services Business Process Execution Language (BPEL) Version 2.0,” OASIS Standard, 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[16] M. Reichert, P. Dadam, and T. Bauer, “Dealing with forward and backward jumps in workflow management systems,” International Journal of Software and Systems Modeling (SOSYM), vol. 2(1), 2003, pp. 37–58.

[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.

[19] W. M. P. van der Aalst, “The application of Petri nets to workflow management,” Journal of Circuits, Systems and Computers, vol. 8(1), 1998, pp. 21–66.

[20] World Wide Web Consortium (W3C), “Web Services Addressing 1.0 – Core,” W3C Recommendation, 2006, http://www.w3.org/TR/ws-addr-core/

[21] Apache Software Foundation, “Apache Orchestration Director Engine (ODE),” http://ode.apache.org/

[22] Eclipse BPEL Project, “Eclipse BPEL Designer,” http://www.eclipse.org/bpel

[23] F. Leymann, “Supporting business transactions via partial backward recovery in workflow management systems,” Proc. of the Conference on Database Systems for Business, Technology and Web (BTW), Springer, 1995.

[24] D. Chiu, Q. Li, and K. Karlapalem, “A meta modeling approach to workflow management systems supporting exception handling,” Journal of Information Systems, Elsevier, vol. 24(2), 1999, pp. 159–184.

[25] Apache Software Foundation, “Failure and Recovery in Apache ODE,” http://ode.apache.org/activity-failure-and-recovery.html

[26] P. Greenfield, A. Fekete, J. Jang, and D. Kuo, “Compensation is not enough,” Proc. of the 7th International Enterprise Distributed Object Computing Conference (EDOC), 2003.

[27] D. Karastoyanova, F. Leymann, and A. Buchmann, “An approach to parameterizing Web service flows,” Proc. of the 3rd International Conference on Service Oriented Computing (ICSOC), 2005, pp. 533–538, doi:10.1007/11596141_45.

[28] 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,” Journal of Nucleic Acids Research, vol. 34, Web Server issue, 2006, pp. 729–732, doi:10.1093/nar/gkl320.

[29] E. Deelman, D. B. Gannon, M. Shields, and I. Taylor, “Workflows and e-Science: An overview of workflow system features and capabilities,” International Journal of Future Generation Computer Systems, Elsevier Science Publishers, vol. 25(5), 2009.pp. 528–540.

[30] I. Altintas, O. Barney, and E. Jaeger-Frank, “Provenance collection support in the Kepler scientific workflow system,” International Provenance and Annotation Workshop (IPAW), Springer, LNCS, vol. 4145, 2006, pp. 118–132.

[31] Z. Luo, “Checkpointing for workflow recovery,” Proc. of the 38th ACM Southeast Regional Conference, 2000, pp. 79–80, doi:10.1145/1127716.1127735.

[32] S. Urban, L. Gao, R. Shrestha, and A. Courter, “Achieving recovery in service composition with assurance points and integration rules (short paper),” Proc. of the OTM Conferences (1), 2010, pp. 428–437.

[33] S. Marzouk, A. J. Maâlej, and M. Jmaiel, “Aspect-oriented checkpointing approach of composed Web services,” Proc. of the 1st Workshop on Engineering SOA and the Web (ESW), Springer, LNCS, vol. 6385, 2010, pp. 301–312.

[34] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. M. P. van der Aalst, “Process flexibility: a survey of contemporary approaches,” Proc. of the 4th International Workshop CIAO! and the 4th International Workshop EOMAS, Springer, LNBIP, vol. 10, 2008, pp. 16–30.

All links were last checked on June 26, 2012.