Model-as-you-go for Choreographies: Rewinding and Repeating Scientific Choreographies

Andreas Weiß[1], Vasilios Andrikopoulos[2], Michael Hahn[1], Dimka Karastoyanova[3]

[1]University of Stuttgart, Germany
{andreas.weiss, michael.hahn}@iaas.uni-stuttgart.de

[2]University of Groningen, The Netherlands
v.andrikopoulos@@rug.nl

[3] Kühne Logistics University (KLU), Hamburg, Germany
dimka.karastoyanova@the-klu.org

Model-as-you-go for Choreographies: Rewinding and Repeating Scientific Choreographies

Andreas Weiß, Vasilios Andrikopoulos, Michael Hahn, and Dimka Karastoyanova

Abstract—Scientists are increasingly using the workflow technology as a means for modeling and execution of scientific experiments. Despite being a very powerful paradigm workflows still lack support for trial-and-error modeling, as well as flexibility mechanisms that enable the ad hoc repetition of experiment logic to enable, for example, the convergence of results or to handle errors. In this respect, in our work on enabling multi-scale/field (multi-*) experiments using choreographies of scientific workflows, we contribute a method comprising all necessary steps to conduct the repetition of choreography logic across all workflow instances participating in a multi-* experiment. To realize the method, we contribute i) a formal model representing choreography models and instances, including the re-execute and iterate operations for choreographies, and based on it ii) algorithms for determining the rewinding points, i.e. the activity instances where the rewinding has to stop and iii) enable the actual rewinding to a previous execution state and repetition of the choreography. We present the implementation of our approach in a message-based, service-oriented system that allows scientists to model, control, and execute scientific choreographies as well as perform the rewinding and repeating of choreography logic. We also provide an evaluation of the performance of our approach.

Index Terms—Ad Hoc Changes, Flexible Choreography, Workflow, Multi-* Experiment, Choreography Rewinding, Choreography Re-execution and Iteration

1 Introduction

Workflow technology offers an approach for the design and implementation of *in-silico experiments* such as scientific simulations. By means of *scientific workflows*, it supports the goal of eScience to provide generic approaches and tools for scientific exploration and discovery in different fields of natural and social sciences [1]. More specifically, scientific workflows are used to specify the control and data flow of in-silico experiments and orchestrate scientific software modules and services. Through the use of workflow technology in eScience, a significant body of knowledge and tools from the business process management domain becomes available to natural scientists. However, at the same time scientists have different requirements on workflow modeling and enactment than users in the business domain. For instance, eScience experiments often demand a trial-and-error based modeling [2] supporting the use of incomplete, partially defined workflow models, or the repetition of the execution of specific experiment steps with different sets of parameters. In this context, natural scientists are both the designers and users of a workflow model.

In order to address these requirements, previous work [3] proposed the *Model-as-you-go* approach for workflows. A key aspect of this approach is that it hides the differences between workflow model and instance by abstracting from technical details such as deployment. This works towards creating the impression of one coherent experimentation phase where scientists can iteratively model and execute a scientific workflow. Model-as-you-go also supports two types of user-initiated operations allowing the *ad hoc repetition* of parts of the workflow model without the a priori definition of execution control artifacts [4]. The *iterate* operation allows repeating workflow logic without undoing previously completed work. This is helpful for scientists to enforce the convergence of results by repeating some steps of the scientific workflow. The *re-execute* operation allows repeating parts of already executed workflow logic after undoing, or compensating already completed work. This allows scientists, for example, to reset the execution environment in case of detected errors or even when a complete redo of a simulation is needed.

A known limitation of this approach, however, is the insufficient support for multi-scale/multi-field, also known as *multi-** experiments, that recent works are addressing [5], [6]. Multi-scale experiments couple different length or time scales within the same experiment, e.g. part of the overall simulation simulates a natural phenomenon on the atomic scale with nanometer length and transfers the results to another simulation application simulating structures with millimeter length. Multi-field experiments use different scientific fields in the same experiment, for example physics, biology, or chemistry. In order to provide better support for these types of in silico experiments, we proposed the use of choreographies [7]. Every scale and every distinct field is modeled as an independent choreography participant. This corresponds to the fact that multi-* experiments typically involve scientists from different disciplines and organizations having diverse expertise [5].

Fig. 1 shows a simplified example of a multi-scale experiment studying the simulation of thermal aging of

- *A. Weiß and M. Hahn are associated with the Institute of Architecture of Applications Systems (IAAS) at the University of Stuttgart, Universitaetsstr. 38, 70569 Stuttgart, Germany*
  E-mail: {firstname.lastname}@iaas.uni-stuttgart.de
- *V. Andrikopoulos is associated with the University of Groningen, Nijenborgh 9, 9747AG Groningen, The Netherlands*
  E-mail: v.andrikopoulos@rug.nl
- *D. Karastoyanova is associated with the Kühne Logistics University (KLU), Großer Grasbrook 17, 20457 Hamburg, Germany*
  E-mail: dimka.karastoyanova@the-klu.org

Manuscript received on May 19, 2017

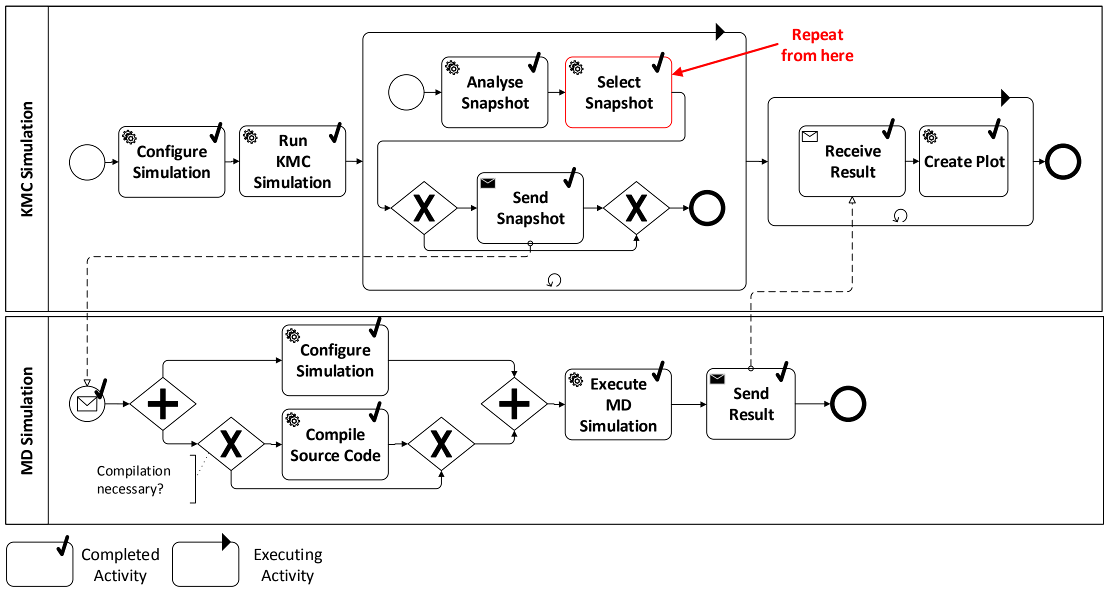


Fig. 1. An example of a choreographed multi-scale simulation of thermal aging of iron-copper alloys and their material behavior. Adapted from [5]

iron-copper alloys and emerging effects on the mechanical behavior of the alloys [8]. The coupling of two simulation methods allows for carrying out simulations on multiple time and length scales. The first method is a Kinetic Monte Carlo (KMC) Simulation and simulates the formation of copper atom clusters (precipitates) in an atom lattice and stores the intermediate results in snapshots at discrete time steps. The generated snapshots are analyzed and sent to a Molecular Dynamics (MD) simulation workflow if the atom clusters have an appropriate size. The MD simulation workflow and the services implementing the activities apply forces on each snapshot to test material behavior after thermal aging. Each received snapshot triggers the creation of a new workflow instance of the MD simulation. The results are sent back to the KMC simulation workflow to generate an overall graphical plot for each simulation snapshot. Together, the two simulation workflows form a choreography with each simulation method being represented by an independent choreography participant.

Let's assume now that a scientist started the coupled simulation workflows and they already have calculated a simulation snapshot, applied forces on the atom lattice, and visualized the result graphically. The scientist discovers that the visualization does not show a plausible graph and wishes to re-run parts of the overall multi-scale simulation. These could be, for example, a change of the criteria that are used to select an appropriate KMC simulation snapshot, the re-sending of snapshots, and the re-run of the MD simulation. Essentially, this requires the capability that allows scientists to select a point in the KMC or MD workflows up to which the execution of the simulations has to be *rewound* before applying any desired changes and repeating the execution of this part of the simulation. Repeating part of the execution instead of discarding all intermediate results and starting from scratch saves a lot of time and effort for the scientists, especially in the case of typically long running experiments.

Toward supporting such type of control over scientific experiments, in this work we build on the concepts we introduced in previous work [4], [5], [6], and present the complete Model-as-you-go for Choreographies approach that allows for rewinding and repetition in choreographies. For this purpose, in Sec. 2 we present a *formal description of choreography models and instances*, which we extended from [6] to incorporate *loop constructs* typically used in acyclic choreography and workflow models. Furthermore, the formal model accounts for the so-called *participant sets* that can be used to model a participant that can be instantiated an arbitrary number of times. Subsequently, in Sec. 3, we introduce a new *method* comprising all steps necessary to rewind and repeat choreography logic, from determining the rewinding points, through rewinding the choreography, enacting a repeat or iteration of its logic, and to resuming the choreography execution. In the same section we present the *algorithm to determine the rewinding points*, which is a combination of its basic version as introduced in [6] and a significant extension to support loop activity instances and instances of participant sets. Rewinding points are activity instances in the participating workflows up to which the state has to be rewound and where the repetition begins. We enable the repetition or iteration of choreography logic by defining and implementing two operations: *iterate* and *re-execute*. For the actual rewinding at the choreography participants, i.e. the resetting or compensation of activities in each involved participant instance, we resort to the concepts of [4], about which we provide the necessary background details. As with the previous works, the concepts introduced in this article are independent of a particular choreography or workflow language and therefore reusable across technologies. Sec. 4 describes how the concepts and algorithms for repetition are realized into a message-based and service-oriented system, the *ChorSystem*, that supports the modeling, execution, and control of scientific choreographies and implements the method we introduced in Sec. 3. In Sec. 5, the proposed approach is experimentally evaluated in terms of performance. Finally, Sec. 6 compares our approach to related ones and Sec. 7 concludes the article.

2 Formal Model

In this section, we define the underlying formal model for our approach in two parts: *modeling* and *execution*.

2.1 Modeling Phase

Typically, choreography models show only the publicly visible communication behavior, because the details of the workflows implementing the choreography participants are considered as sensitive information. The usually non-executable models are defined collaboratively and used to generate representations of the choreography participants in an executable workflow language. The collaborating organizations then refine the resulting workflow models they own with business logic [9].

A choreography model consists of at least two participants, which are represented by service orchestrations, workflow models, or process models. A process model is a directed, acyclic graph (DAG) whose nodes represent activities. Control flow is explicitly modeled by edges denoted as control flow connectors linking activities. Data flow is implicitly described through the manipulation of variables as input and output of activities. The participants communicate with each other via message links, a second type of edges.

A process model is formally defined as in [10]:

Definition 1 (Process Model, G). A *process model* is a DAG $G = (m, V, i, o, A, L)$, where $m \in M$ is the name of the process model, $V \subseteq M \times S$ (M = set of names; S = set of data structures) is the set of variables, i is the map of input variables, o is the map of output variables, A is the set of activities, and L is the set of control flow connectors (control flow links).

The set of activities A contains both basic and loop activities (cf. Definition 2 below). *Input variables* providing data to activities can be assigned using an input variable map $i : A \rightarrow \mathcal{P}(V)$. *Output variables* to which activities may write data to are described by the output variable map $o : A \rightarrow \mathcal{P}(V)$. Finally, the set of control flow connectors is $L \subseteq A \times A \times C$. A *control flow connector* $l \in L$ is a triple $l = (a_s, a_t, t \mid a_s, a_t \in A, t \in C \wedge a_s \neq a_t)$ connecting a source and a target activity, and its *transition condition* t (where C is the set of all conditions) is evaluated during run time. Note that the transition conditions allow to model typical workflow patterns such as parallel split and exclusive choice [11]. An activity $a \in A_*$ is called *start activity* if it is not the target of a control flow link: $A_* \subseteq A := \{a \mid a \in A \wedge \forall l \in L, a \neq \pi_2(l)\}$[1].

Loops in the formal model for process models are expressed as complex activities that execute the loop as a sub-process according to a defined exit condition [10]. More formally:

Definition 2 (Loop Activity, a_l). A *loop activity* $a_l \in A_L$ is a tuple $a_l = (m, A, L, \epsilon)$, where $m \in M$ is the name of the loop activity, A is a set of activities, L is a set of control flow connectors, and ϵ is a function $\epsilon : A_L \rightarrow C$ that assigns an exit condition to a loop activity.

1. Note that we use the projection operator π_n to access the n^{th} element of a tuple starting from index 1. $\mathcal{P}(X)$ denotes the power set of the set X including the empty set $\emptyset$. $p_y.X$ accesses element X (potentially a set) of element p_y.

A loop activity is a container activity for other activities, regular (i.e. non-loop) as well as nested loop activities, and control flow connectors. The loop activity represents a do-until loop, which is executed at least once before the exit condition is evaluated. Our definition of a choreography model integrates the process model tuple as part of its participant definition:

Definition 3 (Choreography Model, $\mathfrak{C}$). A choreography model is a directed, acyclic graph denoted by the tuple $\mathfrak{C} = (m, P, P_{set}, ML)$, where $m \in M$ is the name of the choreography model, P is the set of choreography participants, P_{set} is the set containing participant sets, ML is the set of message links between the choreography participants.

A *choreography participant* $p \in P$ is a triple $p = (m, type, G)$, where $m \in M$ is the name of the participant, $type : P \rightarrow T$ is the function assigning a type $t_p \in T$ to the participant, and $G \in G_{all}$ is a process model graph, where G_{all} is the set of all process model graphs.

Typing the participant allows for several participants of the same type in the same choreography. Participants of the same type always possess the same process model graph. The set of all participants is denoted by P_{all}. A *participant set* $p_{set} \in P_{set}$ is described by $p_{set} \subseteq P_{all}$. This modeling construct is used to model a set of choreography participants whose number can be determined only during run time [12].

The set of message links ML is denoted as $ML \subseteq (P \cup P_{set}) \times P \times A \times A \times C$. A *message link* $ml \in ML$ is a tuple $ml = (p_s, p_r, a_s, a_r, t)$, where p_s, p_r are the sending and receiving participants, which must not be identical: $p_s \neq p_r$. This also holds for $a_s \in \pi_5(p_s.G)$ and $a_r \in \pi_5(p_r.G)$, which are the sending and receiving activities: $a_s \neq a_r$. The transition condition $t \in C$ is evaluated during run time.

2.2 Execution Phase

Choreography models are typically not directly executable [7], [13]. Instead, the refined process/workflow models implementing the choreography participants are instantiated. Together, they form an overall *virtual choreography instance*. The virtual choreography instance at any given point in time can be created by reading monitoring information, i.e. the execution traces of each process instance. We use the definitions of activity and process instance from [4] and extend them for choreography instances. Note that for our purposes it is sufficient to have a rather static notion of a choreography instance capturing only the accrued state at a certain point in time and not the advancement of the execution. This is due to the fact that the execution progress is suspended when we apply our algorithm, as we discuss in the following section.

Definition 4 (Process Instance, p_g). An instance of a process model is a tuple $p_g = (V^I, A^A, A^F, L^E,)$, where V^I is the set of variable instances, A^A is the set of active activity instances, A^F is the set of finished activity instances, and L^E is the set of evaluated links .

For the set of variable instances it holds that: $V^I = \{(v, c, t) \mid v \in V, c \in DOM(v), t \in \mathbb{N}\}$. A variable instance provides a concrete value c from the domain of v ($DOM(v)$) for a variable v at a particular point in time t. The set of activity instances is defined as $A^I = \{(id, a, s, t, \sigma) \mid id \in ID, a \in A, s \in \mathcal{S}, t \in \mathbb{N}, \sigma \in \Sigma\}$, where ID is a set of unique identifiers, A is the set of activities, $\mathcal{S}$ is the set of states, $\mathbb{N}$ is the set of natural numbers indicating time, and Σ is the set of variable snapshot instances. A variable snapshot instance $\sigma \in \Sigma$ is defined as the triple $\sigma = (id, V^I_\sigma, t) \mid id \in ID, V^I_\sigma \subseteq V^I, t \in \mathbb{N}$.

The set $\mathcal{S} = \{S, E, C, F, T, Cmp, D\}$ contains the *execution states* an activity instance can take at any point in time t. Note that we describe states with the following abbreviations: (S=scheduled, E=executing, C=completed, F=faulted, T=terminated, Cmp=compensated, D=dead, Sus=suspended). *Completed* activity instance have been successfully carried out their intended work, while *terminated* ones were forcefully aborted during execution.

The state of an activity instance $a^i \in A^I$ can be determined by the function $state(a^i)$, whereas its model element is retrieved by the function $model(a^i)$. The set of activity instances A^I contains both non-loop and loop (cf. Definition 6) activity instances. For the set of active activity instances the following holds: $A^A \subseteq A^I, \forall a^i \in A^A : state(a^i) \in \{S, E\}$. If an activity is executed in a loop, a new instance is created for each iteration, i.e. there is at most one activity instance a^i of an activity a in the set of active activities A^A. For the set of finished activity instances the following holds: $A^F \subseteq A^I, \forall a^i \in A^F : state(a^i) \in \{C, F, T, D\}$. *Compensated activity instances* (state=Cmp) are not in the set A^F because their effects have been semantically undone. For the set of *evaluated control flow links* the following holds: $L^E = \{(l, c, t) \mid l \in L, c \in \{true, false\}, t \in \mathbb{N}\}$. Evaluated links already have an truth value c assigned at time t determining if the link has been followed during execution. Furthermore:

Definition 5 (Instance Subgraph, g^i). An *instance subgraph* is a directed, acyclic graph represented by a tuple $g^i = (V^I, A^A, A^F, L^E)$, where $g^i \leq_{g^i} p_g$.

The $\leq_{g^i}$ operator means that the elements of the tuple g^i are a subset or equal to the corresponding elements in the process instance tuple p_g. We include the instance subgraph definition in order be able to algorithmically handle the process graph and any subgraphs of it in the same way.

Definition 6 (Loop Activity Instance, a^i_l). A *loop activity instance* is a tuple $a^i_l = (id, a_l, A^A, A^F, L^E, ctr, s, t)$, where $id \in ID$ is a unique identifier of the instance, a_l is the loop activity, A^A is the set of active activity instances, A^F is the set of finished activity instances, L^E is a set of evaluated control flow links, $ctr \in \mathbb{N}$ is the loop counter of the loop activity instance, $s \in \mathcal{S}$ is the current state of the instance, $t \in \mathbb{N}$ is the instance's execution time.

It then follows:

Definition 7 (Choreography Instance, $\mathfrak{c}^i$). A choreography instance is the pair $\mathfrak{c}^i = (P^I, ML^E)$, where P^I is the set of participant instances belonging to the choreography instance and ML^E the set of evaluated message links.

The set of participant instances P^I contains pairs of the form $p^i = (m, p_g)$, where m is the name of the participant instance and $p_g \in P^{all}_g$ is a process instance. For ML^E the following holds: $ML^E = \{(ml, c, t) \mid ml \in ML, c \in \{true, false\}, t \in \mathbb{N}\}$, i.e. ML^E contains the instantiated message links having a truth value c indicating the outcome of the transition condition evaluation at execution time t.

3 Rewinding & Repeating Choreographies

In this section we present our overall approach for rewinding and repeating choreography logic in a choreography instance. We introduce the method our approach follows and the algorithm for automatic identification of rewinding points.

3.1 Prerequisites

A basic assumption for the *Model-as-you-go for Choreographies* approach is the existence of a monitoring infrastructure capturing the execution events and providing information about instance states of the process models distributed across different execution engines. These states are correlated with the corresponding choreography model, both in the supporting middleware and in the graphical modeling and monitoring environment a scientist uses. Based on the life cycle for scientific choreographies, which we introduced in [13], it is possible to switch between the levels of choreography and workflows during execution for monitoring purposes on different degrees of detail and to perform adaptation actions as well starting the repetition on both levels. Each scientist has access to a common choreography model and at least to the workflow models she has refined by herself. This includes the monitoring of the execution state. Access to all other refined workflow models and the execution events depends on the access rights given to the particular scientist by the owner of the workflow model. Furthermore, scientists must be able to control the execution of the choreography instance. Choreography instance control entails the starting, suspending, resuming, and terminating of participating workflow instances in a coordinated fashion in order to be enable to react to deviations and errors, and to perform adaptation actions.

3.2 Concepts & Method

One way to enable reaction to deviations and errors is through repeating workflow/experiment logic using either of two operations we defined for individual (scientific) workflows. *Iteration* is the repetition of parts of the logic in a workflow instance without undoing already completed work. The executed activities and links in the engine internal representation are simply reset; note that historical data are kept and past state changes are not overwritten. Optionally, the scientist may decide to load stored variable snapshots for the re-run and not use the current values. Basically, iteration resembles the execution semantics of a loop construct without having it explicitly modeled. *Re-execution* denotes the operation of repeating workflow logic in a single workflow instance after undoing the already completed work by invoking compensation activities in the reverse order for executed service invocations, resetting control flow links, and by loading stored variable snapshots for the start activity [4]. In the context of (scientific) choreographies, we analogously define *iteration in a choreography instance* as the repetition of logic in the enacting workflow instances without undoing already completed work. The workflow instances participating in the choreography instance are collectively reset. *Re-execution in a choreography instance* is the repeat of logic after compensating already completed work. That means the choreography instance is reset to a state of its

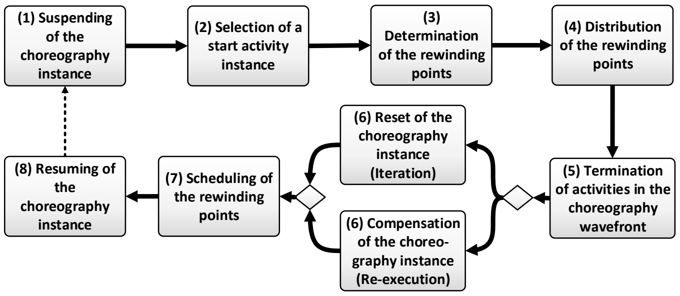

(1) Suspending of the choreography instance
(2) Selection of a start activity instance
(3) Determination of the rewinding points
(4) Distribution of the rewinding points
(5) Termination of activities in the choreography wavefront
(6) Reset of the choreography instance (Iteration)
(6) Compensation of the choreography instance (Re-execution)
(7) Scheduling of the rewinding points
(8) Resuming of the choreography instance

Fig. 2. Method for rewinding and repetition of choreography logic

past. The involved workflow instances have however to be compensated collectively. The rewinding and repetition of choreography logic can be conducted by adhering to the steps of the method depicted in Fig. 2. The steps involved are explained in the following sub-sections making use of several conceptual figures[2] (Fig. 3 - Fig. 5).

3.2.1 Step 1: Suspending of the Choreography Instance

This is done either by a scientist or by the workflow engines reaching the end of the defined workflow models. The user-initiated suspend has to be conducted in a coordinated fashion with all relevant workflow engines by using reliable messaging [14] and/or transactional concepts [15]. It implies that no further execution progress in any of the involved workflow instances occurs until they are collectively resumed again (cf. Step 8).

3.2.2 Step 2: Selection of a Start Activity Instance

Repeating parts of the logic of choreography instances is triggered by the scientist choosing a *start activity* in a *start participant instance* (activity c_1 of $Participant_1$ in Fig. 3) during run time via a graphical modeling and monitoring environment. As multiple (completed) instances for one model element might exist, especially inside loop activities, the modeling and monitoring environment must support a convenient selection, for example by using a graphical wizard that leads the user through the selection step by step. The selection can either take place on the level of the choreography or on the workflow level depending on the access rights of the scientist.

When monitoring the scientific choreography instance, *data*, i.e. the variable values, of the workflow instances is one factor scientists have to consider for the repetition of logic. Similar to database logs for recovery [16], in our existing work [4] we log variable changing events into stable storage. A snapshot-enabled workflow engine stores the *snapshot instances* with every variable-changing activity in its database. Moreover, it offers a service interface to the outside to retrieve the snapshot instances related to any given activity. A snapshot instance contains references to all variables visible for a particular activity, the current variable values, and a creation timestamp. Fig. 4 shows the simplest (i.e. sequential) selection case in a workflow instance. If it is

2. In the following, the subscript of an element denotes a part of the element name, while the superscript indicates the instance id. However, we only show the instance id if there are more than one instances of the same element.

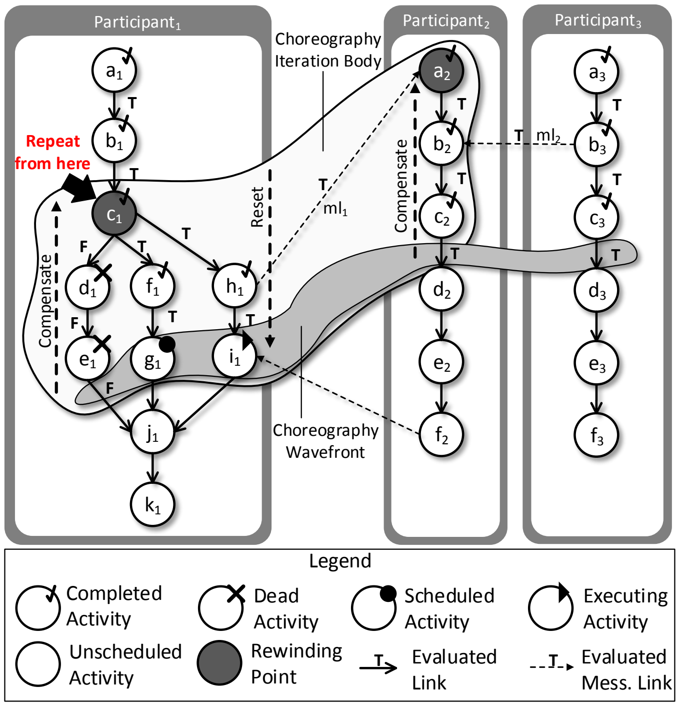


Fig. 3. Example of a choreography instance without loops

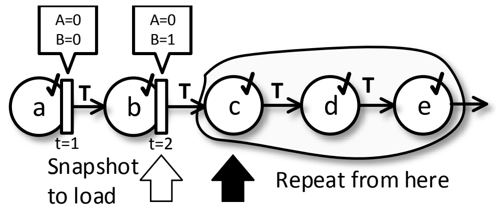


Fig. 4. Data handling in the sequential case. Adapted from [4]

to be repeated from activity instance c, the snapshot instance attached to activity instance b has to be loaded. It is the nearest snapshot instance, i.e. has the most recent timestamp ($t = 2$), on the execution path to activity instance c. The selection of snapshot instances from parallel execution paths are also considered in [4], however, we do not recapitulate these concepts here due to space limitations.

We base our data handling on this concept and allow the user to select either manually or automatically the stored snapshot instance with the most recent timestamp for the start activity instance. This means the corresponding workflow engine is queried for all snapshot instances related to the selected start activity instance and the results are presented to the user in a graphical manner. The option is provided for both iteration and re-execution. In case of re-execution, the compensation logic is only responsible for undoing state which is external to the participant. Therefore, a snapshot instance is has to be selected and loaded to provide the desired variable values, which are part of the internal state of the workflow instance, before starting the re-execution. In case of iteration it is optional.

The refinement of the choreography participants might introduce variables which are not part of the choreography model. Two cases can be distinguished: (i) A scientist wishes to start the repetition of a choreography instance from the choreography or workflow level where he/she has access to the refined workflow model. In this case, graphical facilities are provided to select the desired snapshot instance and all or a subset of the variables visible for the start activity instance. (ii) A scientist wishes to start the repetition on the level of the choreography from a participant instance where he/she does not have access to the refined workflow. Here, the manual selection of snapshots should not be offered. Instead, the engine internally uses the most recent variable snapshot instance preceding the start activity instance from the corresponding workflow engine's database as shown in Fig. 4. The same approach is employed for all calculated rewinding points outside the start participant instance.

3.2.3 Step 3: Determination of the Rewinding Points

An important concept is the notion of *iteration body*. In [4], the iteration body is defined as the instantiated activities and evaluated links reachable from a user-selected start activity instance of a single process instance. In Fig. 3, the iteration body of $Participant_1$ consists of the activities c_1, d_1, e_1,f_1, g_1, h_1, i_1, and the control flow connectors between them. Note that the path $c_1 \rightarrow d_1$ has not been chosen during execution and is marked as *dead*. However, it may be included into the iteration body when starting a repetition from activity c_1. In the context of choreographies, the iteration body spans across process instances and includes message link instances between them. In Fig. 3, in addition to the already enumerated ones, the activities a_2, b_2, c_2, the control flow links between them as well as the control flow link $c_2 \rightarrow d_2$, and the message link instance ml_1 are part of the *choreography iteration body*. In other words, the choreography iteration body contains all activity, control flow, and message link instances reachable from the start activity instance. The repetition of logic starting in one particular participant instance affects at least all participant instances that are part of the choreography iteration body.

Here, two cases can be distinguished. In the first case, the start participant instance is connected, directly or transitively, to other participants that are reachable from the manually selected start activity instance (c_1 in Fig. 3). That means, the start participant instance contains *completed* sending activity instances that have sent messages to other participant instances which themselves may have invoked further ones. Already completed activity instances on the execution path must be *rewound*, i.e. either be reset (iteration) or compensated (re-execution) across the affected choreography participant instances. While the start activity instance for the repetition is simultaneously the so-called *rewinding point* in the start participant instance, the rewinding points in the connected instances have to be identified separately. Each rewinding point indicates where the resetting or compensation of activities has to be stopped. In the example, activity a_2 is the rewinding point of $Participant_2$.

In the second case, participant instances that are not reachable from the start activity instance may still be affected by the repetition of logic in other participant instances. Messages that are not the reply to previous requests and have been transmitted over incoming message links to the affected participant instances must be available again in the case of repetition. This can be done by storing and replaying previously sent messages by the workflow engine or a middleware component responsible for the respective participant instance. An example for this case are $Participant_3$ and the message link instance ml_2 in Fig. 3.

The concept for determining the rewinding points in choreography instances is refined in this work to also consider *multiple instances of the same participant*, i.e. instantiated participant sets (cf. Definition 3), as well as *loop activity instances* as part of the participant instances. Fig. 5a shows an example of a choreography instance containing both loop activity instances and two instances of a choreography participant modeled with the participant set construct. In order to find the rewinding points in all participant instances, all iterations of an instantiated loop activity have to be traversed. The iterations of a loop activity instance can be seen as independent instance subgraphs (cf. Definition 5) where each has to be traversed sequentially. We call this the *loop instance graph* of a particular loop iteration. In the example of Fig. 5a, this means that starting from the initial start activity instance d_1^1 in the first iteration of loop activity instance c_1, the participant instance $Participant_2^1$ is reached via the message link instance ml_1. Activity instance a_2 is the rewinding point of $Participant_2^1$. After finishing the traversal of $Participant_2^1$ and the loop activity instance iteration 1, the loop instance graph of loop activity instance iteration 2 is traversed. Here, $Participant_2^2$ is reached via the message link instance ml_3 and activity instance a_2 can be marked as rewinding point.

Our approach supports two more cases, both shown in Fig. 5b. Firstly, we allow users to select an activity instance in an arbitrary iteration of an loop activity instance. In the example in Fig. 5b this is the activity instance d_1^2 located in iteration 2 of loop activity instance c_1. Secondly, both participant instances possess loops that synchronize with each other, e.g. participant instance $Participant_2$ also possesses a loop activity instance (c_2). Following the message link instance ml_3 to find the rewinding point in $Participant_2$, the correct loop iteration of c_2 must be entered to mark the activity instance d_2^2 as a another rewinding point.

In order to automatically determine the rewinding points, we introduce an algorithm in Sec. 3.3. The data structure to store the rewinding points of a choreography instance is defined in the following way:

Definition 8 (Choreography Rewinding Points, $RP_{\mathfrak{C}}$). The data structure *Choreography Rewinding Points* $RP_{\mathfrak{C}} \subseteq P^I \times \mathcal{P}(A^I)$ is a set of pairs $\{(p^i, A^I_{rp}) \mid p^i \in P^I, A^I_{rp} = \{a^i_1, ..., a^i_k\} \subseteq A^I\}$ consisting of a participant instance and a set of rewinding point activity instances.

The reason that a participant instance can have more than one rewinding point is the existence of parallel paths in the process model graph. A participant may receive messages in parallel that result in independent rewinding points.

3.2.4 Step 4: Distribution of the Rewinding Points

In this step, the automatically determined rewinding points as well as the optionally selected variable snapshot instance (of the start participant instance) have to be distributed to the workflow engines that host the involved workflow instances. It has to be ensured that the rewinding points reach all

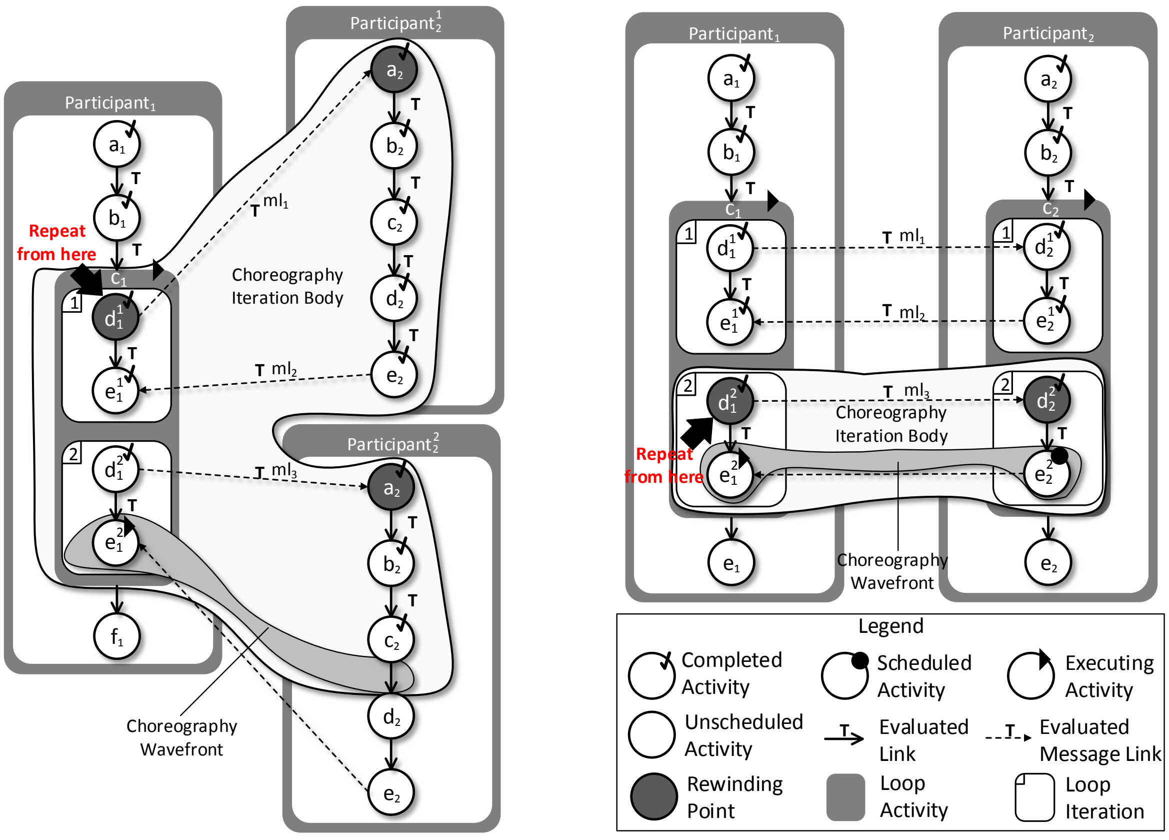


Fig. 5. Loop examples. 5a: Choreography instance with loops and instantiated participant set. 5b: Choreography instance with synchronizing loops

relevant workflow engines, e.g. by using reliable message-oriented middleware [14] and/or transactional concepts [15].

3.2.5 Step 5: Termination of Activities in the Choreography Wavefront

Manually triggered repetition of logic on an execution path can lead to race conditions in individual workflows [4] as well as in choreographies. Race conditions include the execution of one path in parallel inside one process or choreography instance. To deal with this issue the activities of the *choreography wavefront* have to be terminated. The choreography wavefront contains all currently active instances or scheduled elements such as activities, control flow connectors, and message links. In Fig. 3, for example, activities g_1 and i_1 of $Participant_1$ are currently running or are scheduled, respectively, and will subsequently trigger the competing execution of activity j_1 if not terminated before starting the repetition from activity instance c_1. The termination is handled locally in each involved workflow instance after receiving the rewinding points.

3.2.6 Step 6: Rewinding the Choreography Instances

The rewinding resets the choreography iteration body to a previous state and moves the choreography wavefront to the past. This has to be conducted locally by each involved workflow engine. However, different approaches are used for iteration and re-execution (cf. Fig. 2). For the rewinding of each individual workflow instance we reuse the concepts of [4]. That means in the case of iteration that the activities and links in the iteration body of each participant instance are reset by the corresponding workflow engine to enable a subsequent execution. See Fig. 3 for an example that shows the resetting direction from the rewinding point until reaching the choreography wavefront. Optionally, data snapshots are automatically determined for the rewinding points (cf. Step 2).

We formally define the iterate operation of choreography logic in the following way:

Definition 9 (Iterate operation, $\mathfrak{i}_c$). The *iterate operation for choreographies* is defined as the function $\mathfrak{i}_c : A^I \times \Sigma \times RP_{\mathfrak{C}}^{all} \times \mathfrak{C}^I \rightarrow \mathfrak{C}^I$, where A^I is the set of activity instances, Σ is the set of variable snapshot instances, $RP_{\mathfrak{C}}^{all}$ is the set of all choreography rewinding points (cf. Definition 8), and $\mathfrak{C}^I$ is the set of choreography instances.

The iterate operation takes as input the start activity instance $a_{start}^i \in A^I$, the corresponding variable snapshot instance $\sigma \in \Sigma$ of a_{start}^i, the set of determined rewinding points assigned to their corresponding participant instances $RP_{\mathfrak{C}} \in RP_{\mathfrak{C}}^{all}$, and the affected choreography instance $\mathfrak{c}^i \in \mathfrak{C}^I$.

The re-execution operation is an extension of iteration, where additionally to the resetting of control flow connectors, the iteration body of each participant instance is compensated in reverse order (cf. Fig. 3) and a data snapshot is automatically determined for the rewinding points in each involved workflow engine. The semantics of the input parameters is identical to the iterate operation. Note that the re-execution operation is only applicable if corresponding compensation services have been implemented for a particular use case. We also extended the traversal of the iteration body by the workflow engines for both operations to also consider multiple rewinding points.

A special case to be considered is the complete rewinding of a participant instance. This occurs when the model of the rewinding point activity instance is an instance creating activity (for example activity instance a_2 of $Participant_2$ in Fig. 3). Instead of keeping the participant instance alive, it should be terminated since it cannot be guaranteed whether an instance of that particular participant is needed again during the repetition. The new execution might take a completely different path. However, simply terminating the participant instance is not sufficient. In case of re-execution the iteration body still has to be compensated to undo its effects. In case of iteration, the iteration body has to be reset and the variable values have to be stored. If the repeated execution takes the same path, the variable values have to be loaded for a new participant instance of the same type. However, if the participant instance is created on a different workflow engine, the variable values have to be migrated.

3.2.7 Step 7: Scheduling of the Rewinding Points

The rewinding points are scheduled in their respective workflow engines to be executed next. An exception are the rewinding points that are located in participant instances that have been terminated after rewinding.

3.2.8 Step 8: Resuming of the Choreography Instance

The last step entails the resuming of the execution of the choreography instance. To each participant instance, with the exception of the completely rewound and terminated ones, a message is sent to resume execution.

The method's steps can be repeated if a scientist recognizes the need for further repetitions. Here, no new challenges for the determining of the rewinding points arise. Furthermore, if one step of the method fails, all steps apart from the compensation of activities in case of re-execution can be easily retried by starting the method anew. This is due to fact that the workflow engines keep their instance state in stable storage and the calculation of the rewinding points and resetting already reset activities does not do any damage. However, retrying the compensation step would need a idempotent implementation of external services providing the compensation functionality.

3.3 Determining the Rewinding Points

In the following, we present an algorithm to automatically determine the rewinding points in a choreography iteration body, that realizes Step 3 of the proposed method. The main idea of the algorithm is to traverse the choreography instance graph beginning from the user-selected start activity instance in the start participant instance, follow the executed message link instances and thus identify all choreography participant instances that are part of the choreography iteration body. In doing so, the algorithm collects all rewinding points. The algorithm is divided into four parts and supported by a set of auxiliary functions, which are defined and explained as they occur in the algorithm. The rewinding point algorithm realizes the function ρ.

Algorithm 1: determineRewindingPoints, ρ

```
input  : Choreography instance $\mathfrak{c}^i$, activity instance $a^i_{start}$,
         participant instance $p^i$, set of pairs $RP_{\mathfrak{C}} = (p^i, A^I_{rp})$
output : $RP_{\mathfrak{C}}$
begin
    if $RP_{\mathfrak{C}} = \emptyset$ then
        $RP_{\mathfrak{C}} \leftarrow (p^i, \{a^i_{start}\})$
    end
    Loop Activity Instance
      $a^i_l \leftarrow$ getEnclosingLoop($a^i_{start}$)
    if $a^i_l = \perp$ then
        $\tau\ (c^i, p^i, p^i.p_g, a^i_{start}, RP_{\mathfrak{C}})$
    else
        $\lambda\ (\mathfrak{c}^i, p^i, a^i_l, a^i_{start}, RP_{\mathfrak{C}})$
    end
    return $RP_{\mathfrak{C}}$
end
```

Algorithm 2: traverseInstanceSubgraph, τ

```
input  : Choreography instance $\mathfrak{c}^i$, participant instance $p^i$,
         instance subgraph $g^i$, activity instance $a^i_{start}$, set
         of pairs $RP_{\mathfrak{C}} = (p^i, A^I_{rp})$
output : $RP_{\mathfrak{C}}$
begin
    Stack $S \leftarrow \emptyset$
    S.push($a^i_{start}$)
    while $S \neq \emptyset$ do
        $a^i \leftarrow$ S.pop()
        if $a^i$ is not marked as visited then
            mark $a^i$ as visited
            if model($a^i$) is sending activity $\wedge$
              $state(a^i) = completed$ then
                $RP_{\mathfrak{C}} \leftarrow \chi\ (\mathfrak{c}^i, a^i, RP_{\mathfrak{C}})$
                foreach $l^i = (l^x, c^x, t^x) \in g^i.L^E \mid l^x.a^x_s =$
                  $a^i \wedge state(c^x) = true$ do
                    S.push($l^x.a^x_t$)
                end
            else if model($a^i$) is loop activity then
                $RP_{\mathfrak{C}} \leftarrow \lambda\ (\mathfrak{c}^i, p^i, a^i, \perp, RP_{\mathfrak{C}})$
            end
        end
    end
    return $RP_{\mathfrak{C}}$
end
```

Definition 10 (Function ρ). The *Determine Rewinding Points function* ρ is defined as $\rho : \mathfrak{C}^I \times A^I \times P^I \times RP^{all}_{\mathfrak{C}} \rightarrow RP^{all}_{\mathfrak{C}}$, where $\mathfrak{C}^I$ is the set of choreography instances, A^I is the set of activity instances, P^I is the set of participant instances, and $RP^{all}_{\mathfrak{C}}$ is the set that contains all $RP_{\mathfrak{C}}$ sets (cf. Definition 8).

$RP_{\mathfrak{C}}$ is the data structure that contains the determined rewinding points for each involved participant instance. Before the first invocation of ρ the data structure $RP_{\mathfrak{C}}$ is empty: $RP_{\mathfrak{C}} = \emptyset$. Algorithm 1 is the starting point for the automatic determination of the rewinding points. The initial invocation of the algorithm needs the start participant instance and the user-selected start activity instance. First, it is checked if the start activity instance a^i_{start} is nested inside a loop activity instance using the *getEnclosingLoop* function (cf. Definition 11 below). If so, the user has selected an activity instance inside an iteration of a loop activity instance and the sub-routine λ (*handleLoopActivity*) defined in Definition 15 and realized by Algorithm 4 is directly invoked. Otherwise, a^i_{start} is not nested inside a loop and the function τ (*traverseInstanceSubgraph*) is invoked to traverse the process instance graph p_g of p^i.

Definition 11 (Function getEnclosingLoop). The *getEnclosingLoop function* is defined as $getEnclosingLoop : A^I \rightarrow A^I_L$, where A^I is the set of activity instances and A^I_L is the set of loop activity instances.

The *getEnclosingLoop* function is used to determine if there is a loop activity instance $a^i_l \in A^I_L$ enclosing a given activity instance $a^i \in A^I$. It returns $\perp$ if a^i does not have a parent loop activity instance, and function τ is invoked:

Definition 12 (Function τ). The *Traverse Instance Subgraph τ function* is defined as $\tau : \mathfrak{C}^I \times P^I \times G^I \times A^I \times RP^{all}_{\mathfrak{C}} \rightarrow RP^{all}_{\mathfrak{C}}$, where $\mathfrak{C}^I$ is the set of choreography instances, P^I is the set of participant instances, G^I is the set of instance subgraphs, A^I is the set of activity instances, and $RP^{all}_{\mathfrak{C}}$ is the set that contains all $RP_{\mathfrak{C}}$ sets (cf. Definition 8).

The function τ is used to traverse an instance (sub-)graph. It is realized by Algorithm 2, the main idea being the following: beginning from the activity instance a^i_{start}, the instance subgraph g^i is traversed in a depth-first manner. g^i can be the complete process instance graph p_g or a loop instance graph of some iteration of a loop activity instance. For each activity instance a^i it is checked if it is a completed sending activity instance (line 10). If so, the sub-routine χ (*handleSendingActivity*) as defined in Definition 13 is invoked (line 11). Subsequently, the outgoing control flow connectors of every completed activity instance are followed by pushing them on the stack data structure, provided they have been evaluated to true (lines 12-14). If the model of a^i ($model(a^i)$) is a loop activity, the sub-routine λ (*handleLoopActivity*, cf. Definition 15 and Algorithm 4) is invoked instead (lines 15-17).

Definition 13 (Function χ). The *Handle Sending Activity function* χ is defined as $\chi : \mathfrak{C}^I \times A^I \times RP^{all}_{\mathfrak{C}} \rightarrow RP^{all}_{\mathfrak{C}}$, where $\mathfrak{C}^I$ is the set of choreography instances, A^I is the set of activity instances, and $RP^{all}_{\mathfrak{C}}$ is the set of pairs containing the assignment of participant instances to their rewinding points ($RP_{\mathfrak{C}}$, cf. Definition 8).

Algorithm 3 (handleSendingActivity) realizes the function χ. First, the message link instance $ml^i_{traversed}$, which is attached to the sending activity instance a^i, is retrieved by evaluating the following conditions: (i) it has been evaluated to true and (ii) there is a *receiving* activity instance a^i_r in the *completed* state, i.e. a message has been sent and consumed (line 4). We assume reliable FIFO channels for communication, i.e. all messages in transit have reached their destination before we conduct any rewinding. If $ml^i_{traversed}$ is not empty, the algorithm follows the message link instance and retrieves the receiving participant instance (lines 5-6). By exactly identifying the receiving participant instance it is also possible to consider instances which were modeled by a participant set. For the receiving participant instance it is checked if it has already been (partly) traversed by the algorithm and a (preliminary) rewinding point has been found. If this is not the case, the receiving activity instance a^i_r is added as a rewinding point for the receiving participant p^i_r and ρ is invoked recursively using p^i_r as input (lines 7-9). If there exists already a rewinding point for p^i_r (line 10), it is checked if (i) the old rewinding point would be a successor of the new one (using the *succ* function defined in Definition 14 below) or if (ii) both are in parallel branches. In case (i) the old rewinding point activity instance is removed before the new rewinding point is added and in case (ii) both are kept (lines 11-19). In both cases, ρ is invoked recursively afterwards (lines 20-23). The recursion in one participant instance stops when all reachable completed activity instances have been marked as visited.

Algorithm 3: handleSendingActivity, χ

```
1  input  : Choreography instance $\mathfrak{c}^i$, activity instance $a^i$, set
            of pairs $RP_{\mathfrak{C}} = (p^i, A^I_{rp})$
2  output : $RP_{\mathfrak{C}}$
3  begin
4      Message Link Instance $ml^i_{traversed} \leftarrow (ml^x, c^x, t^x) \mid$
         $(ml^x, c^x, t^x) \in ML^E \wedge ml^x = (p^i_s, p^i_r, a^i_s, a^i_r, c) \wedge a^i =$
         $a^i_s \wedge state(c^x) = true \wedge state(a^i_r) = completed$
5      if $ml^i_{traversed} \neq \perp$ then
6          Participant Instance $p^i_r \leftarrow ml^i_{traversed}.p^i_r$
7          if $\nexists (p^x, A^x_{rp}) \in RP_{\mathfrak{C}} \mid p^x = p^i_r$ then
8              $RP_{\mathfrak{C}} \leftarrow RP_{\mathfrak{C}} \cup (p^i_r, \{a^i_r\})$
9              $RP_{\mathfrak{C}} \leftarrow \rho\ (\mathfrak{c}^i, a^i_r, p^i_r, RP_{\mathfrak{C}})$
10         else if $\exists (p^x, A^x_{rp}) \in RP_{\mathfrak{C}} \mid p^x = p^i_r$ then
11             Boolean $recursion \leftarrow false$
12             foreach $a^x \in A^x_{rp}$ do
13                 if succ($a^i_r, a^x$) then
14                     $A^x_{rp} \leftarrow A^x_{rp} \setminus a^x$
15                     $recursion \leftarrow true$
16                 else if $\neg$succ($a^x, a^i_r$) $\wedge$ $\neg$succ($a^i_r, a^x$) then
17                     $recursion \leftarrow true$
18                 end
19             end
20             if recursion then
21                 $A^x_{rp} \leftarrow A^x_{rp} \cup a^i_r$
22                 $RP_{\mathfrak{C}} \leftarrow \rho\ (\mathfrak{c}^i, a^i_r, p^i_r, RP_{\mathfrak{C}})$
23             end
24         end
25     end
26     return $RP_{\mathfrak{C}}$
27 end
```

Definition 14 (Function succ). The *successor function succ* is defined as $succ : A^I \times A^I \rightarrow \mathbb{B}$, where A^I is the set of activity instances and $\mathbb{B}$ is the set of boolean values $\mathbb{B} = \{true, false\}$.

The function determines if the second activity instance is reachable from the first activity instance, and thus is a successor in the process instance graph. When determining the successor property, the iterations of loop activity instances must also be considered.

The handling of loop activity instances and their iterations is introduced using the λ function.

Definition 15 (Function λ). The *Handle Loop Activity* function λ is defined as $\lambda : \mathfrak{C}^I \times P^I \times A^I_L \times A^I \times RP^{all}_{\mathfrak{C}} \rightarrow RP^{all}_{\mathfrak{C}}$, where P^I is the set of participant instances, A^I_L is the set of loop activity instances, A^I is the set of activity instances, and $RP^{all}_{\mathfrak{C}}$ is the set of pairs containing the assignment of participant instances to their rewinding points ($RP_{\mathfrak{C}}$, cf. Definition 8).

Definition 16 (Function getLoopIteration). The *getLoopIteration* function is defined as $getLoopIteration : A^I \rightarrow \mathbb{N}$, where A^I is the set of activity instances and $\mathbb{N}$ is the set of natural numbers.

This function is used to determine the loop iteration of the loop activity instance where a particular activity instance $a^i \in A^I$ is located in.

Definition 17 (Function getLoopInstanceGraph). The *getLoopInstanceGraph* function is defined as $getLoopInstanceGraph : A^I_L \times \mathbb{N} \rightarrow G^I$, where A^I_L is the set of loop activity instances, $\mathbb{N}$ is the set of natural numbers, and G^I is the set of instance subgraphs.

This function is used to retrieve the instance subgraph $g^i \in G^I$ corresponding to a particular loop iteration $ctr \in \mathbb{N}$ of an loop activity instance $a^i_l \in A^I_L$.

The λ function's realization (as required by Algorithm 1) is introduced in Algorithm 4. If there exists a start activity instance $a^i_{start} \in a^i_l.A^A \cup a^i_l.A^F$ nested in the loop activity instance a^i_l, the iteration number $iter_a$ is retrieved using the *getLoopIteration* (cf. Definition 16) function (lines 6-9). Otherwise, $iter_a$ has been initialized to 1 and all loop iterations are traversed. The algorithm iterates while the currently iteration $iter_{curr}$ is smaller or equal to the overall number of iterations of a^i_l indicated by the loop counter $a^i_l.ctr$ (line 10). For each executed iteration, the instance subgraph of the current loop iteration is retrieved using the *getLoopInstanceGraph* (cf. Definition 17) function (line 11). If a^i_{start} exists and it is the first traversal of the loop activity instance ($iter_a = iter_{curr}$), the sub-routine τ (traverseInstanceGraph) is directly called. That means, the traversal does not start at the beginning of the loop instance graph, but rather from a^i_{start} (lines 12-13). Otherwise, the traversal comprises the complete loop instance graph g^i and τ is called for each activity instance a^i, which is a start activity, i.e. $model(a^i) \in A_*$ (cf. Definition 1) (lines 15-17). After the traversal of g^i, $iter_{curr}$ is incremented by 1.

Algorithm 4: handleLoopActivity, λ

```
1  input  : Choreography instance $\mathfrak{c}^i$, participant instance $p^i$,
            activity loop instance $a^i_l$, activity instance $a^i_{start}$,
            set of pairs $RP_{\mathfrak{C}} = (p^i, A^I_{rp})$
2  output : $RP_{\mathfrak{C}}$
3  begin
4      Number $iter_{curr} \in \mathbb{N} \leftarrow 1$
5      Number $iter_a \in \mathbb{N} \leftarrow 1$
6      if $a^i_{start} \neq \perp$ then
7          $iter_a \leftarrow$ getLoopIteration($a^i_{start}$)
8          $iter_{curr} \leftarrow iter_a$
9      end
10     while $iter_{curr} \leq a^i_l.ctr$ do
11         Instance Subgraph
             $g^i \leftarrow$ getLoopInstanceGraph($a^i_l, iter_{curr}$)
12         if $a^i_{start} \neq \perp \wedge iter_a = iter_{curr}$ then
13             $RP_{\mathfrak{C}} \leftarrow \tau(\mathfrak{c}^i, p^i, g^i, a^i_{start}, RP_{\mathfrak{C}})$
14         else
15             foreach $a^i \in g^i.A^F \mid model(a^i) \in A_*$ do
16                 $RP_{\mathfrak{C}} \leftarrow \tau(\mathfrak{c}^i, p^i, g^i, a^i, RP_{\mathfrak{C}})$
17             end
18         end
19         $iter_{curr} \leftarrow iter_{curr} + 1$
20     end
21     return $RP_{\mathfrak{C}}$
22 end
```

4 Realization

In the following, we show how our approach for rewinding and repeating of choreography logic is realized in our *ChorSystem*. The service-oriented and message-based ChorSystem enables users to manage the complete life cycle of choreographies from modeling to execution [17]. The life cycle starts with choreography modeling, transformation to and refinement of workflow models. The refined workflow models are distributed among a set of workflow engines in an automated manner while a logical representation of the choreography is created in the so-called *ChorSystem Middleware*. When starting a new choreography instance, a corresponding representation is created and updated by monitoring execution events. The life cycle management operations for suspending, resuming, and terminating act upon this representation. The functionality of the middleware is defined by composing the different components using message-based Enterprise Integration Patterns [14]. The generic architecture and the control and deployment aspects of choreography life cycle management have been initially introduced in [17]. Due to space limitations we will not discuss the complete system architecture and functionalities in detail and will restrict ourselves to showing with the help of Fig. 6 how the rewinding and repetition method (cf. Fig. 2) is supported by the ChorSystem.

In the *ChorDesigner*, which acts as integrated modeling and monitoring environment as well as control panel for the life cycle management operations, the user gives the command to suspend a running choreography instance (Method step 1). The suspend command is processed by the *Instance Manager* in the so-called *Control Route* [17]. To realize the repetitions the Instance Manager carries out the *Repetition Route* depicted in Fig. 6. The selection of the start activity instance and the corresponding variable snapshot (step 2) are conducted together with the ChorDesigner. This information is sent via messaging to the *Instance manager* where the repetition functionality is found using the *Content-Based Router* pattern. Step 3 of the method is realized by retrieving a choreography instance representation from the Event Registry using a *Custom Message Processor* and subsequently calculating the rewinding point in the second *Custom Message Processor* implementing the Algorithm introduced in Sec. 3.3. In step 4 with the help of the *Recipient List* pattern, the affected workflow engines are determined by accessing the *Management Registry* and translating the repetition message into the workflow engine specific format and adding the calculated rewinding points. The messages are then fanned out to the workflow engines. The reset or compensation of the involved workflow instances (step 6) and the scheduling of the rewinding points (step 7) are carried out in the respective workflow engines as described in Sec. 3 and in [4]. Resuming execution of the choreography instance is again triggered by the user in the ChorDesigner.

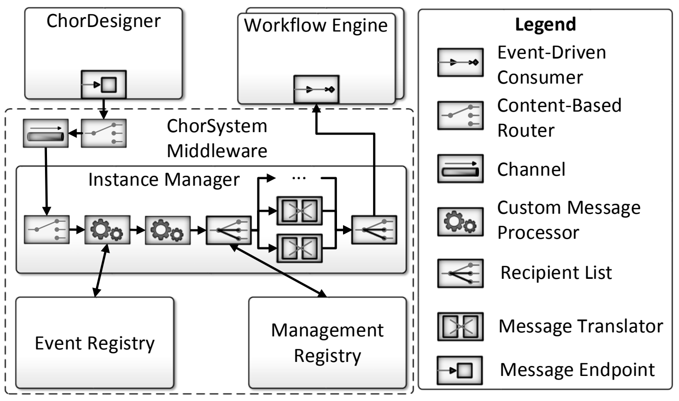


Fig. 6. Repetition Route described by the Enterprise Integration Patterns

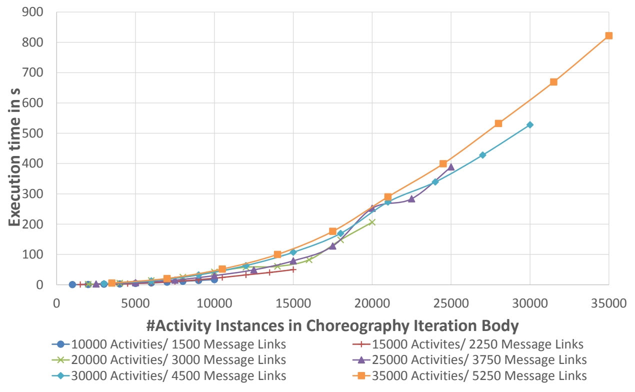
We have implemented the architecture and our approach using open source software and standards. We employ BPEL4Chor [18] as choreography language and BPEL [19] as executable workflow language. The ChorDesigner and the ProcessDesigner are based on Eclipse[3] technologies, while for the ESB Apache ServiceMix[4] is employed. The workflow engines are based on an extended Apache ODE[5], and the message routes to coordinate the middleware components are realized with Apache Camel[6]. The algorithm for finding the rewinding points is implemented in Java. The ChorSystem Middleware's source code is available on GitHub[7]. Detailed information on the ChorSystem and its architecture can be accessed online on our project website [20].

5 EVALUATION

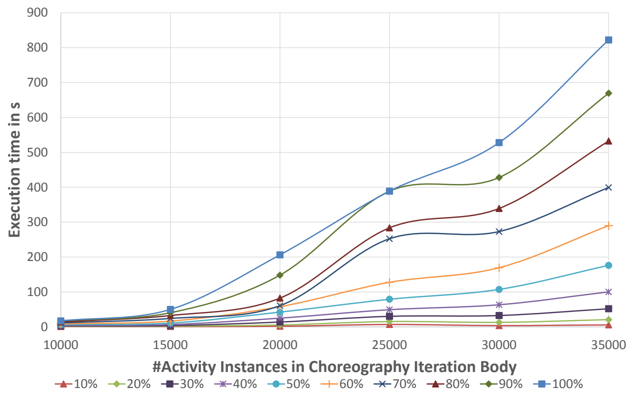
In this section, we evaluate the performance of the proposed algorithm for the rewinding of choreography instances in terms of execution time. The focus in this article on the algorithmical aspect only is due to space reasons. The interested reader is referred to the evaluation of our prototypical ChorSystem by means of an case study, which is available online [20]. All measurements were conducted on an Ubuntu 14.04 LTS virtual machine equipped with 1 CPU core (2.29 GHz) and 4 GB RAM and represent the median values of 5 runs. The evaluation consists of three major parts.

First, we generated (randomly) 6 choreography models having 10 participants each. The reason for generating artificial models is that the execution of our motivating example (cf. Fig. 1) yields execution times of the rewinding algorithm that are too small for meaningful measurements (in the range of milliseconds). Each model increases linearly in size in terms of included activities (from 10K to 35K) and message links between the participants (from 1.5K to 5.25K). Subsequently, we generated for each choreography model 10 iteration bodies with varying wavefronts and increasing size of executed activity instances and traversed message link instances. Fig. 7 summarizes our findings in terms of execution time for the 6 generated choreography models. As

3. http://eclipse.org/modeling
4. http://servicemix.apache.org
5. http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/
6. http://camel.apache.org
7. https://github.com/chorsystem/middleware



(a) Absolute size

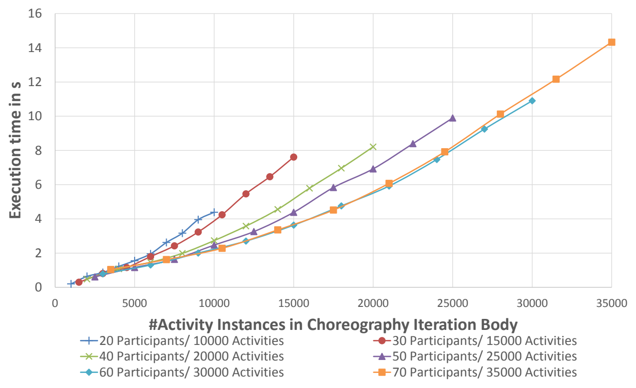


(b) Relative size

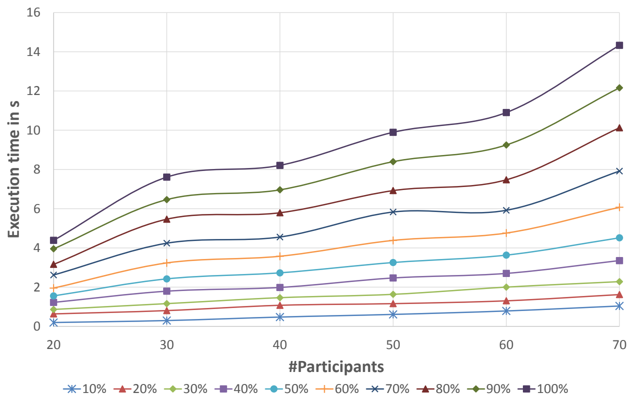
Fig. 7. Execution time (in seconds) for choreography models with constant number of participants and varying size of activities and message links

can be seen from Fig. 7a, the execution time of the rewinding algorithm is quadratic with the number of generated activity and message link instances in the iteration body of each individual model. Likewise, the comparison of the execution time between different models for the same relative size of iteration body, e.g. 50% of included activity and message link instances, shows a quadratic increase of execution time (Fig. 7b). The quadratic growth of the execution time with the number of included activity and message link instances can be explained by looking at Algorithms 2 (traverseInstanceGraph) and 3 (handleSendingActivities). If *no loops* are present, Algorithm 2 traverses each participant instance graph exactly once ($O(n)$, for all participants), while the *succ* function used in Algorithm 3 also traverses the complete participant instance in the worst case. This results in $O(n \times n)$ for each participant instance in the choreography model, as confirmed by Fig. 7a. and 7b.

The second part of the performance evaluation consists of the random generation of 6 new choreography models having 500 activities per participant, but increasing linearly in size in terms of participants (20 to 70) and message links (1.5K to 5.25K). Again, 10 different iteration bodies were generated for each model using the same approach as described above. Fig. 8 shows the execution time for the 10 generated iteration bodies of all 6 choreography models. As before, the observed execution time growth is quadratic with the model size (Fig. 8a). However, the overall execution time is much smaller compared to the models with increasing number of activities per participant. This can be explained

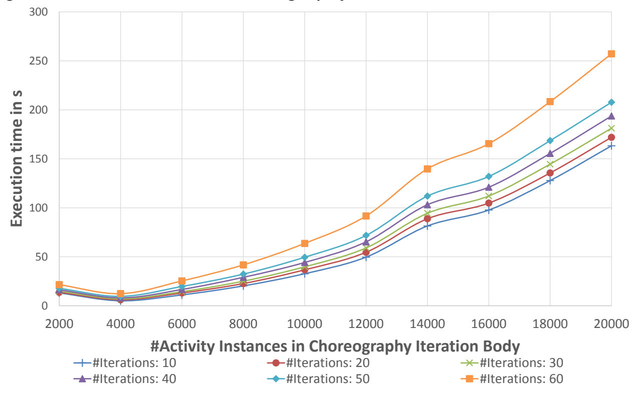


(a) Absolute size

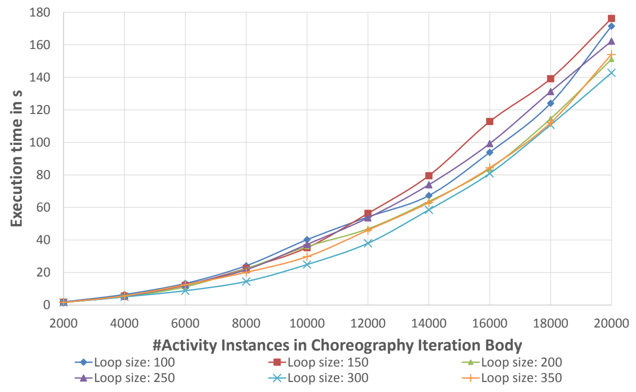


(b) Relative size

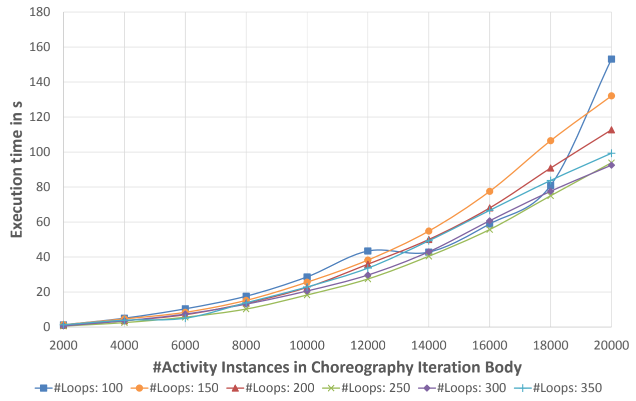
Fig. 8. Execution time of choreography models with constant number of activities and varying size of participants and message links



(a) #Loops=100, Loop Size=3, Increasing #Iterations



(b) #Loops=1, #Iterations=10, Increasing Loop Size



(c) Loop Size=3, #Iterations=10, Increasing #Loops

Fig. 9. Execution time (in seconds) of choreography models with loops (20K Activities, 3K Message Links)

by the fact that the *succ* function (cf. Definition 14) has to traverse a much smaller amount of activity instances per participant, thus, yielding a faster execution time. This explanation is also supported by the execution times of the iteration bodies with the same relative size across the 6 generated models (Fig. 8b), which show a linear increase.

In the third part, in order to evaluate the impact of loops to our approach, we conducted three different measurements varying one of three parameters: number of loops per participant, number of activities per loop (loop size), and number of iterations per loop. For each of the three measurements, we generated a model having 10 participants with 2K activities each and 3K message links between them. Again, for each of the three measurements, 10 choreography iteration bodies of increasing size were generated. If the generated choreography wavefront lies inside a loop activity, we always simulate the worst case scenario where all specified iterations of the corresponding loop activity have been executed. Fig. 9 summarizes the findings of the conducted measurements. In general, all measurements confirm the quadratic increase of execution time. Fig. 9a shows execution times when varying the number of iterations per loop. This increase between the graphs is induced by the increasing number of activity instances that need to be traversed with each additional loop iteration. Fig. 9b shows that the loop activity itself does not influence the execution time when only increasing the loop size parameter per participant. Fig. 9c shows the measurements when only increasing the number of loops per participant. Again, the loop activities itself do not add any significant overhead to the execution time and its growth is also quadratic. We can therefore conclude that the presence of loops in the choreography models, while adding to the overall execution time due to an increased number of activity instances to be traversed, does not significantly affect the performance of our algorithm.

6 Related Work

There are several areas related to our work, such as ad hoc repetition in process instances, rollback-recovery and log-based protocols, and algorithms for consistent global state and predicate detection in distributed systems. In literature, the concept of ad hoc repetition in process instances is well studied. For example, in [21] concepts and algorithms for premodeled or ad hoc backward jumps, which enable the repeat of logic in process instances enacted by the ADEPT system are presented. The Kepler system supports the concept of smart re-runs [22] enabling scientists to repeat parts of a scientific workflow with a different set of parameters. Previously stored provenance information is used to avoid the repetition of parts of the workflow that do not change the overall outcome of the scientific experiment. In [23] process flexibility approaches are studied and classified by type. Our concept for rewinding and repeating choreography instances could be classified as *Flexibility by Deviation* – deviating from the specified control flow in the model. Similarly, our repetition is one form of the *Support for Instance-Specific Changes* as described in [24] for individual process instances. However, none of the above mentioned works consider choreography instances and the implications of messages sent between the participant instances.

Our approach also bears some resemblance to rollback-recovery and log-based protocols facilitating distributed state restoration in message passing systems [25], [26]. These approaches provide means to reset a system of communicating processes to a rollback point in the execution of a program in case of failures. Failures may occur in any participating process and have an effect on other processes in the system due to passed messages. The proposed protocols then identify rollback points either by using log information or by sending checkpoint information. However, there are major differences between this family of protocols and our approach. Firstly, we operate one a much higher level with regard to the employed languages. While the distributed protocols simply assume a set of communicating low level processes of some program execution, our approach operates on complex choreography and workflow instances. They are explicitly described by corresponding language constructs. These include the support for parallel execution inside one participant. Secondly, the intent and scopes of the approaches are quite different. The rollback-recovery protocols are triggered automatically in case of failures. While our approach can be used to react to failures, it is rather meant as a means of user-driven control of a choreography instance during execution to flexibly react to events. Furthermore, we do not only support the restoration to a previous state using the re-execute operation but rather also enable the repeated execution of logic with the iterate operation resembling an enforced loop without it being explicitly modeled. Apart from the language level the mentioned arguments also apply for more recent approaches [27], [28], [29] for checkpointing and rollback-recovery for composite web services/workflows.

With regard to robustness and reliability of workflow executions, [30] proposes a pattern-based approach for specifying transactional properties of service compositions. Furthermore, [31] provides an approach for mining logs of transactional workflows in order to improve the original model. While we ensure robustness by allowing the user to actively influence the execution of a choreography instance as necessary for explorative modeling and execution in domains such as eScience, these approaches focus more on the reliability by design. However, a combination of both approaches seems promising.

Another class of algorithms possessing similarities to our approach are algorithms for observing consistent global states and predicate detection. For example, Chandy and Lamport [32] introduce an algorithm for determining a snapshot of global state in a distributed system in order to detect stable predicates such as termination. However, finding a rewinding point in a choreography instance would not be possible with this kind of algorithm as it can be seen as a unstable predicate detection problem. Marzullo and Neiger [33] present one of the first algorithms using a centralized monitor to record state changes of all processes in a distributed system in order to evaluate if certain unstable predicates hold during execution. This is done by constructing state lattices. The problem of predicate detection in general is NP-complete. However, more efficient solutions for restricted scenarios exist [34]. In our approach, we are also able to achieve an efficient solution by choosing a graph-based data structure fitting our purposes.

7 Conclusions and Future Work

In this article, we motivated the need for the capability to repeat partially or completely the logic in a choreography instance with a clear focus on the eScience community. Toward this goal, we presented a formal model describing choreography models and instances while also considering loops and multiple instances of a particular participant. Based on the formal model, we introduced the concept of repeating logic in choreography instances, which we also expressed through the steps of a corresponding method. We distinguish between iteration, which executes logic again without undoing already completed work, and re-execution, which aims at the compensation of already completed work before executing it again. We defined an algorithm that is able to automatically identify the rewinding points for each involved participant instance. Furthermore, we presented a system for execution and monitoring of choreography instances that supports the proposed concepts and method. The rewinding algorithm has been experimentally evaluated in terms of performance and shown to be acceptable for the purposes of the application domain.

In future, we plan to evaluate our approach and the supporting system in cooperation with other groups of scientists in the context of the SimTech project[8]. In addition, we will work towards enabling transparent data provisioning among participants in flexible choreographies in a manner decoupled from the actual choreography conversations.

Acknowledgments

This work is funded by the project FP7 EU-FET 600792 ALLOW Ensembles and the German DFG within the Cluster of Excellence (EXC310/2) SimTech. Furthermore, the authors would like to thank David R. Schäfer for the fruitful discussions during the writing of this article.

References

[1] T. Hey, S. Tansley, and K. Tolle, Eds., *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, 2009.

[2] R. Barga and D. Gannon, "Scientific versus Business Workflows," in *Workflows for e-Science*. Springer, 2007, pp. 9–16.

8. http://www.simtech.uni-stuttgart.de

[3] M. Sonntag and D. Karastoyanova, "Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows," *Grid Computing*, vol. 11, no. 3, pp. 553–583, 2013.
[4] ——, "Ad hoc Iteration and Re-execution of Activities in Workflows," *Int. J. On Adv. in Softw.*, vol. 5, no. 1 & 2, pp. 91–109, 2012.
[5] A. Weiß and D. Karastoyanova, "Enabling coupled multi-scale, multi-field experiments through choreographies of data-driven scientific simulations," *Computing*, vol. 98, no. 4, pp. 439–467, 2016.
[6] A. Weiß, V. Andrikopoulos, M. Hahn, and D. Karastoyanova, "Rewinding and Repeating Scientific Choreographies," in *CoopIS'15*. Springer, 2015, pp. 337–347.
[7] G. Decker, O. Kopp, and A. Barros, "An Introduction to Service Choreographies," *Inf. Technology*, vol. 50, no. 2, pp. 122–127, 2008.
[8] D. Molnar, R. Mukherjee, A. Choudhury, A. Mora, P. Binkele, M. Selzer, B. Nestler, and S. Schmauder, "Multiscale simulations on the coarsening of cu-rich precipitates in a-fe using kinetic monte carlo, molecular dynamics and phase-field simulations," *Acta Materialia*, vol. 60, no. 20, pp. 6961–6971, 2012.
[9] W. van der Aalst and M. Weske, "The P2P Approach to Interorganizational Workflows," in *CAiSE'01*. Springer, 2001, pp. 140–156.
[10] F. Leymann and D. Roller, *Production Workflow - Concepts and Techniques*. PTR Prentice Hall, 2000.
[11] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow patterns," *Dist. and Par. Databases*, vol. 14, no. 1, pp. 5–51, 2003.
[12] G. Decker, O. Kopp, F. Leymann, and M. Weske, "BPEL4Chor: Extending BPEL for Modeling Choreographies," in *ICWS '07*. IEEE, 2007, pp. 296–303.
[13] A. Weiß and D. Karastoyanova, "A Life Cycle for Coupled Multi-Scale, Multi-Field Experiments Realized through Choreographies," in *EDOC'14*. IEEE, 2014, pp. 234–241.
[14] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.
[15] P. Newcomer and E. Bernstein, in *Principles of Transaction Processing*, second edition ed. Morgan Kaufmann, 2009.
[16] P. A. Berstein, V. Hadzilacos, and N. Goodman, *Concurreny Control And Recovery in DataData Systems*. Addison-Wesley, 1987.
[17] A. Weiß, V. Andrikopoulos, S. Gómez Sáez, M. Hahn, and D. Karastoyanova, "ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies," in *CoopIS'16*. Springer, 2016, pp. 503–521.
[18] G. Decker, O. Kopp, F. Leymann, and M. Weske, "Interacting services: from specification to execution," *Data Knowl. Eng.*, vol. 68, no. 10, pp. 946–972, 2009.
[19] OASIS, "Web Services Business Process Execution Language Version 2.0," 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
[20] A. Weiß. Chorsystem project website. [Online]. Available: http://www.iaas.uni-stuttgart.de/chorsystem/
[21] M. Reichert, P. Dadam, and T. Bauer, "Dealing with forward and backward jumps in workflow management systems," *Software and Systems Modeling*, vol. 2, no. 1, pp. 37–58, 2003.
[22] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance Collection Support in the Kepler Scientific Workflow System," in *Provenance and Annotation of Data*. Springer, 2006, pp. 118–132.
[23] H. Schonenberg, R. Mans, N. Russell, N. Mulyar, and W. van der Aalst, "Process flexibility: A survey of contemporary approaches," in *Adv. in Enterprise Eng. I*. Springer, 2008, vol. 10, pp. 16–30.
[24] B. Weber, M. Reichert, and S. Rinderle-Ma, "Change patterns and change support features - enhancing flexibility in process-aware information systems," *Data Knowl. Eng.*, vol. 66, no. 3, pp. 438–466, 2008.
[25] D. L. Russell, "State Restoration in Systems of Communicating Processes," *IEEE Softw. Eng.*, vol. SE-6, no. 2, pp. 183–194, 1980.
[26] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A Survey of Rollback-recovery Protocols in Message-passing Systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.
[27] H. Mansour and T. Dillon, "Dependability and Rollback Recovery for Composite Web Services," *IEEE Transactions on Services Computing*, vol. 4, no. 4, pp. 328–339, 2011.
[28] S. D. Urban, L. Gao, R. Shrestha, and A. Courter, "Achieving Recovery in Service Composition with Assurance Points and Integration Rules," in *OTM'10*. Springer, 2010, pp. 428–437.
[29] Y. Xiao and S. Urban, "Using Rules and Data Dependencies for the Recovery of Concurrent Processes in a Service-Oriented Environment," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 59–71, 2012.
[30] S. Bhiri, W. Gaaloul, C. Godart, O. Perrin, M. Zaremba, and W. Derguech, "Ensuring customised transactional reliability of composite services," *J. of Database Man.*, vol. 22, no. 2, pp. 64–92, 2011.
[31] W. Gaaloul, K. Gaaloul, S. Bhiri, A. Haller, and M. Hauswirth, "Log-based transactional workflow mining," *Dist. and Par. Databases*, vol. 25, no. 3, pp. 193–240, 2009.
[32] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.
[33] K. Marzullo and G. Neiger, "Detection of global state predicates," in *Int. Workshop on Dist. Algorithms*. Springer, 1991, pp. 254–272.
[34] C. M. Chase and V. K. Garg, "Detection of global predicates: Techniques and their limitations," *Distributed Computing*, vol. 11, no. 4, pp. 191–201, 1998.

Andreas Weiß received the M.Sc. degree in Business Information Systems in 2013 from the University of Stuttgart and the University of Hohenheim, Stuttgart, Germany. Since then, he is working as a research associate at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart. His main topics of research interest lay in the flexible modeling and execution of choreographies for scientific use cases such as simulations.

Vasilios Andrikopoulos is an assistant professor of software engineering at the University of Groningen. His research is in the area of software architectures for service-oriented, cloud-based, and hybrid systems and infrastructures, as well as software engineering with an emphasis on evolution and adaptation. He received his PhD from Tilburg University, the Netherlands. He has experience in research and teaching Database Systems and Management, Software Modeling and Programming, Business Process Management and Integration, and Service Engineering. He has participated in a number of EU projects, including the Network of Excellence S-Cube.

Michael Hahn received the Dipl.-Inf. degree in Computer Science from the University of Stuttgart in 2013. Currently, he works as a research associate and PhD student at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart. His research interests are Service Oriented Computing and Architectures, focusing on the optimization of data exchange between choreographed services based on use cases from the e-Science domain.

Dimka Karastoyanova is an associate professor of data science and business intelligence at the Kühne Logistics University. Her research and teaching activities are in the fields of service-oriented computing, BPM and flexible workflow management, data science, and Cloud computing, and in application domains like eScience, supply chain management and logistics. She received her PhD in Computer Science from Technische Universität Darmstadt, Germany. She was on the research and management team of European Projects like the NoE S-Cube, IP SUPER and FET ALLOWEnsambles, and of the German Cluster of Excellence SimTech.