ChorSystem: A Message-based System for the Life Cycle Management of Choreographies

Andreas Weiß[1], Vasilios Andrikopoulos[1], Santiago Gómez Sáez[1], Michael Hahn[1], Dimka Karastoyanova[2]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{andreas.weiss, vasilios.andrikopoulos, santiago.gomez-saez, michael.hahn}@iaas.uni-stuttgart.de

[2]Kühne Logistics University, Hamburg, Germany
Dimka.Karastoyanova@the-klu.org

ChorSystem: A Message-based System for the Life Cycle Management of Choreographies

Andreas Weiß[1], Vasilios Andrikopoulos[1], Santiago Gómez Sáez[1],
Michael Hahn[1], and Dimka Karastoyanova[2]

[1]Institute of Architecture of Application Systems (IAAS)
University of Stuttgart, Germany
{andreas.weiss,vasilios.andrikopoulos,santiago.gomez-saez,
michael.hahn}@iaas.uni-stuttgart.de
[2]Kühne Logistics University, Hamburg, Germany
Dimka.Karastoyanova@the-klu.org

Abstract. Service choreographies are commonly used as the means for enabling inter-organizational collaboration by providing a global view on the message exchange between involved participants. Choreographies are ideal for a number of application domains that are classified under the Collaborative, Dynamic & Complex (CDC) systems area. System users in these application domains require facilities to control the execution of a choreography instance such as suspending, resuming or terminating, and thus actively control its life cycle. We support this requirement by introducing the ChorSystem, a system capable of managing the complete life cycle of choreographies from choreography modeling, through deployment, to execution and monitoring. The performance evaluation of the life cycle operations shows that the ChorSystem introduces an acceptable performance overhead compared to purely script-based scenarios, while gaining the abilities to control the choreography life cycle.

Keywords: Collaborative Dynamic Complex (CDC) Systems, Choreography Life Cycle Management, Flexible Choreographies

1 Introduction

The notion of Collaborative, Dynamic & Complex (CDC) systems [4] aims to unify the concepts and tools of several application domains which exhibit overlapping requirements but use diverging solutions to address them. Examples of these domains include eScience, i.e. scientific workflows, pervasive adaptive systems, and service networks. CDC systems' operation is best described by a three-phase life cycle consisting of the phases of Modeling, Provisioning, and Execution. In order to model the communication behavior between different applications and services in CDC systems, we rely on the concept of choreographies [9]. Choreographies, also known as *inter-organizational workflows*, are used in the business domain to enable the collaboration of independent organizations toward a common goal. A choreography model provides a global view on the message exchange between the involved parties denoted by choreography participants. Choreographies do not have a centralized party coordinating the message exchange, but rather act in a distributed peer-to-peer-like manner. Since choreography models are typically not directly executable, the choreography participants are transformed into an executable workflow representation and refined by adding additional business logic [8]. During execution (or run time) the instances of the refined workflows form an overall choreography instance, which can be (re-)constructed by reading the choreography related execution events of each workflow engine [22].

In literature, various aspects of the choreography life cycle are discussed, from modeling [1, 9] through design time compliance checking [13], to execution and monitoring in order to check the run time validity of choreography instances to their models [17, 25]. However, these concepts only cover some parts of the CDC system life cycle. While there is detailed work for the CDC *Provisioning* phase that allows to provision workflow middleware and its underlying infrastructure [20], to the best of our knowledge only [11] briefly discusses the deployment of service choreographies, however, without elaborating on its realization. Furthermore, there are only a few works on controlling the collective execution of a set of workflow instances forming a choreography instance [14, 17]. One reason for this is the assumption that the constituent workflow models are kept private by their owning organizations and that no (logically) centralized party has access to them [1]. Instead, the refined workflow models constituting a choreography are to be deployed by each responsible party. Furthermore, no collaborating party might have the authority to influence the execution of the choreography instance. In our view, this is a valid assumption in general and especially in traditional inter-organizational settings. However, the application domains under the CDC umbrella, such as Collective Adaptive Systems [3] or eScience [24] demand facilities that go beyond only triggering and monitoring the execution of a set of workflows forming a choreography. What is needed are operations that enable the users to actively influence and control the execution of these workflow instances as a part of the management of the overall choreography life cycle.

We would like to illustrate this with the help of Fig. 1 depicting a choreography example from the eScience application domain. The figure shows the high level workflow steps of a material science simulation studying the thermal aging of iron-copper alloys by coupling two distinct simulation methods [15]. Thermal aging occurs when steel is subject to high pressures and temperatures altering the steel's physical properties and leading, for example, to crack formations. Simulating thermal aging allows better predictions about material behavior. The first method in the example is a Kinetic Monte Carlo (KMC) Simulation, which simulates the formation of copper precipitates in an atom lattice and stores the intermediate results in snapshots at particular 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. The results are sent back to the KMC simulation workflow

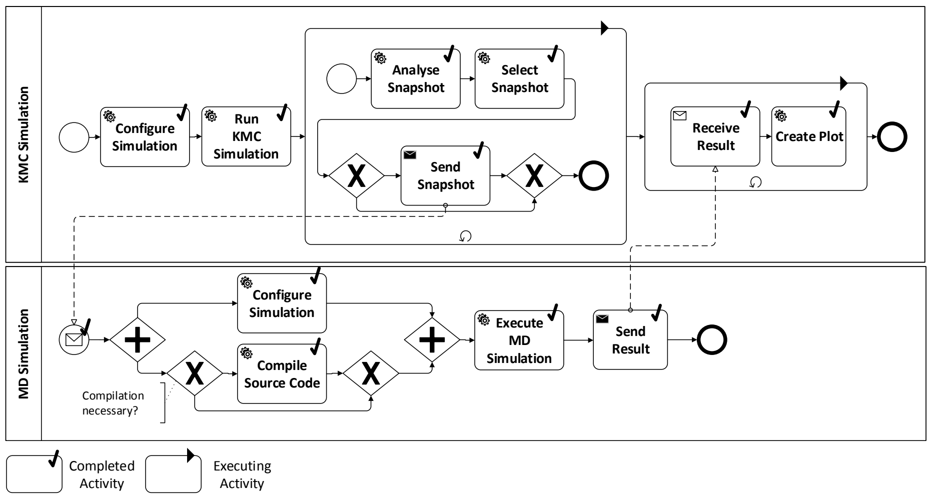


Fig. 1: Scientific choreography example. Adapted from [22]

to generate an overall graphical plot for each simulation snapshot. Together, the two simulation workflows form a choreography.

In order to provide flexibility in coupled scenarios as the one described above and to support the explorative nature of scientific discovery [6], we introduced the notion of *Model-as-you-go for Choreographies* [24]. It allows users to adapt choreographies after the execution of the implementing workflows has already been started. We described the underlying approach by a choreography life cycle where the user can seamlessly switch between modeling and execution phases [23]. Furthermore, we discussed how choreography logic can be reset to a previous execution state for error handling or to repeat parts of the execution in [22]. For example, a scientist might have started the coupled simulation workflows and let their execution proceed to the depicted state in Fig. 1. The scientist discovers that the visualization does not show a plausible graph and wishes to re-run parts of the overall simulation. This could be a change of the criteria that are used to select an appropriate KMC simulation snapshot (*Select Snapshot* activity), the re-sending of snapshots, and the re-run of the MD simulation. Repeating the execution instead of discarding all intermediate results saves a lot of time especially in case of the typically long running scientific experiments.

However, in all our previous works, we only mentioned very briefly that users need the ability to deploy and control a choreography as a prerequisite, but did not present any details how this can be achieved. In order to be able to apply any changes to a running choreography, as discussed in the context of Fig. 1, the following challenges have to be addressed:

(i) How can all artifacts implementing a choreography model be collectively deployed onto a distributed execution environment by non-IT experts such as scientists? In our example, the artifacts would be the two workflow models as well as associated interface descriptions.

(ii) How can users start, monitor, and control the execution of a choreography? In our example, scientists must be able to recognize any undesired results, pause the execution of all workflow instances collectively before applying any changes (see [22] for details on this) as well as resume the workflow instances afterwards. This type of control must be provided to the user in an easy and intuitive manner, preferably via graphical tools hiding the underlying complexity.

Our paper addresses these open challenges by focusing on the question on what kind of instrumentation is required so that the Provisioning and Execution phase of CDC systems can be supported. More specifically, we discuss the automatic and transparent deployment of the workflows, which implement choreography participants, to a set of workflow engines, and the subsequent instantiation of a choreography model. Furthermore, we argue that choreography life cycle management entails more than just triggering the execution of individual workflow instances, but also needs a logical representation of the choreography instance state upon which life cycle operations can be enacted. Toward this goal, we first discuss in further detail the phases of the choreography life cycle, and identify a set of corresponding requirements for a choreography life cycle management middleware (Section 2). Section 3 introduces the ChorSystem, our novel choreography life cycle management system, which relies on the well-known Enterprise Integration Patterns (EIP) [12] for its realization. We then detail our approach for automating the deployment of choreographies and for controlling choreography instances. Additionally, we give some insights about the ChorSystem's prototypical implementation. Section 4 describes the performance evaluation we conducted for the middleware part of the ChorSystem using a realistic case study and illustrate how the life cycle operations are triggered by the system's users. We position our work with regard to related work in Section 5 and conclude with a summary and outlook to future work in Section 6.

2 Choreography Life Cycle Management

We identify the following operations for life cycle management of choreographies: *model*, *transform*, *refine*, *deploy*, *start*, *suspend*, *resume*, *terminate*, and *undeploy*. Modeling a choreography, transforming it into as set of workflow models, and refining them with additional logic is discussed further in our previous work, see for example [24]. The operations deploy, start, and undeploy are enacted on a choreography model and its related artifacts required for its instantiation. Suspend, resume, and terminate are enacted upon choreography instances.

The deployment of a choreography and all related artifacts such as the implementing workflow models is a prerequisite for all other life cycle management operations. It enables the creation of a logical representation of the choreography and monitoring its global state, based on which all other life cycle operations can be enacted. Furthermore, automated deployment of the refined workflow models reduces time and effort, especially in distributed execution across multiple nodes scenarios. However, to address privacy requirements, we see the deployment of choreographies as a two-stage operation:

(i) registering each choreography participant and the interfaces it offers in a logically centralized, but physically distributed middleware such as an Enterprise Service Bus [7]
(ii) the actual deployment of all participant related artifacts onto one or more workflow engines

If privacy is an issue, only step (i) can be conducted for the affected participant; the actual deployment is then triggered by the responsible party and only its outcome must be conveyed to the middleware in form of an event message.

Automation of the deployment could potentially be achieved by means of scripts invoking whatever deployment interface a workflow engine provides. However, if the global state of the choreography has to be captured in order to let other scripts act on this state information in order to *control* the choreography instance, one basically ends up emulating some kind of middleware layer. Moreover, the choreography users often are non-IT experts as in the CDC application domain of eScience. Therefore, appropriate abstractions are needed for deployment and life cycle management, ideally in a graphical manner and integrated with the modeling environment. This also favors a middleware-centric approach over the use of scripts. Another argument for introducing a middleware layer, and especially when using asynchronous, message-based concepts, is that it provides processing reliability [12]. The following requirements can be identified for a choreography life cycle management middleware based on the above:

R1. **Transparent deployment/undeployment**: The deployment of the refined workflow models implementing choreographies should be automated if privacy requirements do not forbid it. Ideally, users should not worry about placing their refined workflows manually onto the workflow engines. Depending on the privacy requirements of the users, the workflow models should be either placed on private workflow engines which they have registered at the system, or on a set of generally available workflow engines known to the system. The same degree of transparency should apply for undeployment.
R2. **Choreography instantiation**: The system should provide functionality to instantiate a choreography model. This includes the convenient entry of initial parameters for the choreography.
R3. **Monitoring capabilities**: The system should offer facilities to monitor the choreography instance on different levels of detail. This includes, on the one hand, monitoring on the level of the collaboratively defined choreography model and on the other hand, on the level of the refined workflow models a particular user has access to.
R4. **Choreography control**: The system should enable users to actively control the execution of a choreography instance (i.e. support the suspend, resume, and terminate operations).

R5. **Model-as-you-go for Choreographies support**: The notion of Model-as-you-go for Choreographies allows users to flexibly model and execute choreographies, e.g. by rewinding already executed choreography logic and running it again [22]. The system should support these concepts.
R6. **Transparent message routing**: The workflow engines participating in a choreography instance should not be required to keep track of other workflow engines. The message routing between the clients/users and engines should be handled transparently by the system.
R7. **Modeling tool integration** The required facilities of the requirements R1-R5 should be integrated with the graphical modeling tool the user employs in order to conveniently interact with the system and hide its complexities especially for non-IT experts.

3 The ChorSystem

In this section, we first provide an overview of the ChorSystem architecture that fulfills the requirements we identified in the previous section. Subsequently, we present the operations and components involved in the deployment and control of choreographies and give details on the system's implementation.

3.1 Architecture Overview

Figure 2 shows the architecture of the *ChorSystem*. It consists of the Modeling and Monitoring Environment, the ChorSystem Middleware components, an Enterprise Service Bus (ESB) as its messaging backbone, and a set of workflow engines.

ChorSystem Modeling and Monitoring Environment: The *ChorSystem Modeling and Monitoring Environment* comprises three components. The *ChorDesigner* is the graphical tool for modeling choreographies and serves also as the control panel for the deployment of choreographies (see Section 3.2) as well as for starting and controlling a new choreography instance (see Section 3.3). Choreography models are transformed into a set of executable workflow models by the *Transformer* component. The workflow models can then be refined with the workflow modeling tool, the *ProcessDesigner*. One of the major goals of our work is to integrate modeling with execution and monitoring and thus support requirements R3 (*monitoring capabilities*) and R7 (*modeling tool integration*). These requirements have been successfully fulfilled in our previous work for individual workflows by publishing execution events to a message topic which the ProcessDesigner can subscribe to [19]. The received events are then correlated with elements of the workflow model which are colored according to their state in their graphical representation in the modeling tool. This allows detailed monitoring on the level of the workflow models. To enable the monitoring of choreography instances, we use again this concept: the ChorDesigner subscribes to a particular topic in the ChorSystem Middleware where all involved workflow engines publish their choreography model related execution events. The events of

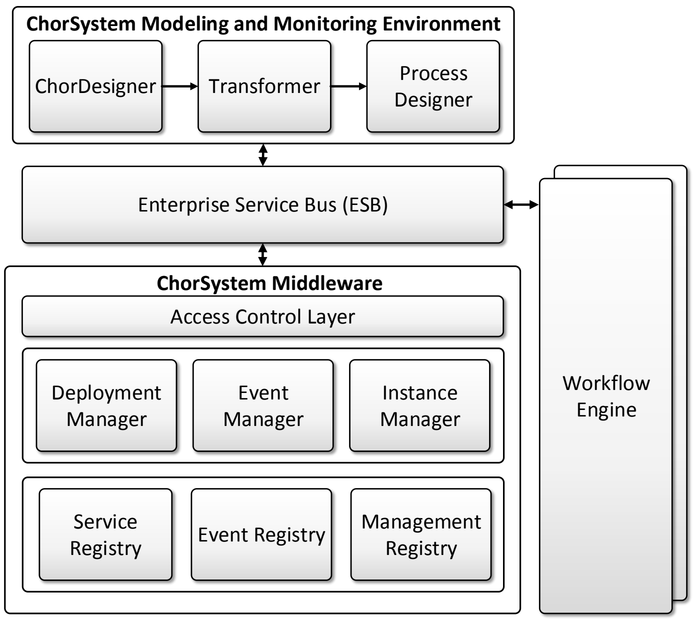


Fig. 2: Architecture of the ChorSystem

a choreography instance identified by a globally unique identifier are correlated with the elements of the corresponding choreography model and then colored for monitoring purposes.

Enterprise Service Bus: An *Enterprise Service Bus* provides the messaging backbone for the ChorSystem and connects the ChorSystem Modeling and Monitoring Environment with the *workflow engines*. It is used as platform where the ChorSystem Middleware components can be plugged in [7] and provided as a service. Although the ESB appears to be only one component in Fig. 2, the ESB concept as described by Chappell [7] allows the physical distribution and clustering of the ESB while still appearing as one logically centralized component. We adopt this notion for our approach. Moreover, we do not violate against the decentralized choreography notion because the actual execution of the choreography participants still happens on independent workflow engines. The use of an ESB helps to satisfy requirement R6 (*transparent message routing*). The goal of this requirement is to remove the burden from each workflow engine participating in a choreography instance to know the location of other workflow engines. We address this by using well-known functional service discovery and selection concepts [16] and WS-Addressing[1] for asynchronous callbacks.

ChorSystem Middleware: The *ChorSystem Middleware* houses all components that are necessary for registering and deploying choreographies as well as

[1] `https://www.w3.org/TR/ws-addr-core/`

instantiating and controlling choreography instances. The *Registry* components act as data services for the *Manager* components. The Manager components are realized by message routes that compose all functionality and are expressed using the Enterprise Integration Patterns [12]. The *Deployment Manager* component is responsible for registering logical representations of the choreography model and the corresponding workflow models in the *Event Registry.* The *Event Manager* is able to update their state during deployment and run time by listening to the execution events published by the workflow engines, and calculating composite events standing for choreography instance state changes. The Deployment Manager optionally distributes the process bundles among the workflow engines. All interfaces offered by a workflow model are registered in the *Service Registry.* This information is used during execution to enable message routing between the workflow engines that have no knowledge of each other. The *Access Control Layer* supervises the access of the users to the choreography and workflow model representations as well as to the execution events.

One of the most important components is the *Instance Manager.* It is responsible for distributing the start messages for instantiating a choreography instance to the workflow engines hosting the respective workflow models. It also creates a corresponding choreography instance representation, which is stored in the Event Registry. Furthermore, the Instance Manager distributes the life cycle control messages among the workflow engines and performs the discovery and selection of services. The Instance Manager is the component where all the functionality for the rewinding and repetition of choreography logic [22] is located and thus fulfills requirement R5 (*Model-as-you-go for Choreographies support*). This includes the creation of a choreography instance graph by correlating the choreography model with its execution events and the implementation of the algorithm for determining the rewinding points. The *Management Registry* is where information about the available workflow engines in the system and their management endpoints are kept. The registration of workflow engines with the middleware is done via an API, for either manual or automated registration.

3.2 Automated Choreography Deployment

In order to allow the automated deployment of refined workflows implementing choreography participants onto a set of workflow engines – requirement R1 (*transparent deployment/undeployment*) – we introduce (i) the concept of a so-called *choreography bundle* and (ii) define a *choreography deployment message route* using the Enterprise Integration Patterns [12].

A *choreography bundle* is a package that physically contains all necessary artifacts for registering a choreography with the ChorSystem Middleware and deploying it onto a set of workflow engines if desired. All meta-data about a choreography bundle and its artifacts are described by a *choreography deployment descriptor.* The descriptor is generated during the transformation step from the choreography model to the implementing workflow models and can be manually enriched during workflow refinement. Figure 3 shows the choreography deployment descriptor meta-model. A choreography bundle consists of a set of *process bundles*,

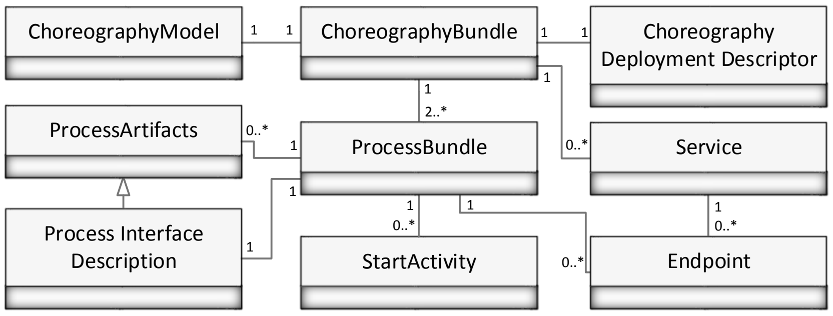


Fig. 3: Meta-model of the choreography deployment descriptor

where each process bundle contains all artifacts necessary to register and deploy one particular workflow model implementing a choreography participant. More precisely, these artifacts are the service descriptions specifying the services the workflow exposes (in the process interface description) and interacts with. If privacy constraints do not prohibit disclosure, the process bundle also contains the actual refined workflow model (possibly encrypted) and platform-specific workflow deployment descriptors. Additionally, the choreography bundle contains the choreography model and the choreography deployment descriptor itself. The choreography deployment descriptor may also codify information about already provisioned non-participant services that need to be registered in the context of the choreography. This information relates to the corresponding service interfaces and endpoints. Moreover, the choreography deployment descriptor may point to the endpoints of already registered workflow engines in the system if the user does not want a particular process bundle to be deployed on any arbitrary engine but rather on one he/she controls. That implies that we offer the possibility to register private workflow engines in the system. The information specified in the choreography deployment descriptor is used to build a *choreography deployment message*, which includes the process bundles and the specified artifacts. The deployment message is processed in the ChorSystem Middleware to register and (if permitted) distribute the process bundles to the registered workflow engines.

With respect to implementation of the deploy operation, the *message route to deploy* a choreography bundle is presented in Fig. 4. The choreography deployment is started from the ChorDesigner by packaging the choreography bundle inside a deployment message and sending it to the ChorSystem Middleware, where it is routed to the Deployment Manager using the *Content-Based Router* pattern. The Deployment Managers' functionality is coordinated by the message route. The first step is to evaluate if there is already a representation of the choreography model registered in the Event Registry using logic implemented in a custom message processor in the route. If not, such a representation is generated. The same holds for the workflow models for which the deployment status is also checked. Each undeployed process bundle is marked accordingly in the deployment message, which is routed to the next message processor implementing the *Splitter* pattern.

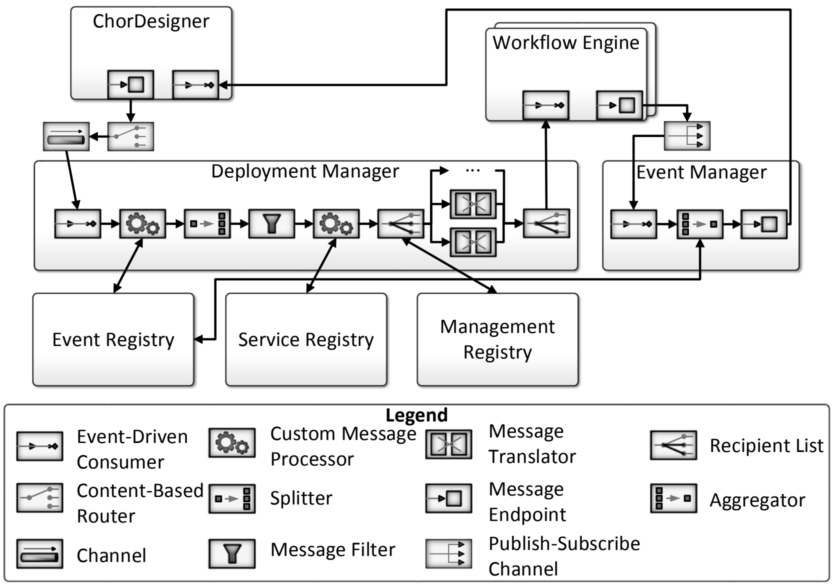


Fig. 4: Deployment Route described using the Enterprise Integration Patterns

Here, the deployment message is split into independent messages containing the process bundles. The messages are subsequently filtered (using the *Message Filter* pattern) and only the ones containing undeployed process bundles are routed in parallel to a message processor. There, the service interfaces a particular workflow model offers are registered in the Service Registry to use them for service discovery and message routing during execution. Subsequently, using the *Recipient List* pattern, the list of workflow engine deployment endpoints is retrieved from the Management Registry. Depending on the information configured in the choreography deployment descriptor either the given workflow engines or automatically chosen ones are retrieved for this task. In the latter case, the middleware layer might randomly chose appropriate workflow engines or consider other factors such as load if available. If a particular process bundle has to be deployed on different types of workflow engines, the process deployment message is cloned accordingly and routed in parallel to a message processor implementing the *Message Translator* pattern, transforming the deployment messages into the format the deployment endpoint of the particular type of workflow engine understands. Finally, the transformed deployment messages are fanned out in parallel to the selected workflow engine endpoints using again a *Recipient List*.

Also visible in Fig. 4 is the event route which is part of the Event Manager. It listens to a *Publish-Subscribe Channel* to which the deployment event messages of the workflow engines are published and updates the workflow model representations in the Event Registry. The deployment event messages are aggregated

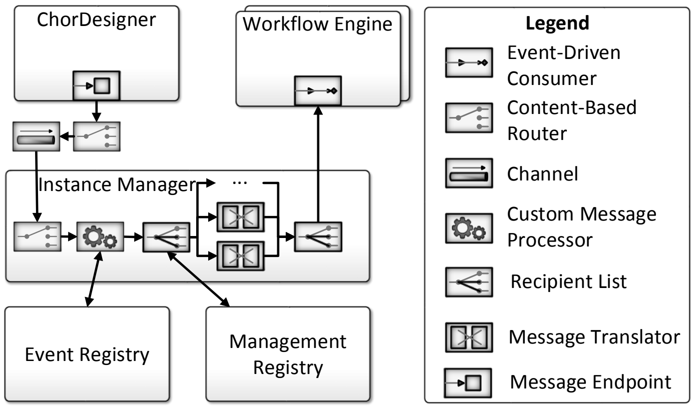


Fig. 5: Control Route described using the Enterprise Integration Patterns

(*Aggregator* pattern) and the ChorDesigner is notified with a corresponding aggregated event message after all workflow models have been successfully deployed.

3.3 Choreography Control

Figure 5 shows the *Control Route* composing the functionality of the Instance Manager. The Control Route distributes the control messages realizing the post-deploy life cycle management operations to the workflow engines, hence supporting requirement R4 (*choreography control*). First, a control message coming from the ChorDesigner is routed to the Instance Manager via a *Content-Based Router*, where the respective operation is triggered using the same pattern. Subsequently, the representation of the choreography instance is retrieved from the Event Registry using custom logic in the route. The *Recipient List* pattern is used to route the control message to the *Message Translator* processors for each involved workflow engine. The type of the workflow engine is retrieved from the *Management Registry* in this step. After translating the messages to the format expected by the target workflow engines, a second *Recipient List* is employed to clone and send the translated control messages for each target workflow engine.

The choreography start message and the subsequent creation of a choreography instance is a special case of choreography control. An empty start message structure is generated by reading the start activity information located in the choreography deployment descriptor (see Fig. 3). The message structures are filled manually by the user or employed to generate appropriate GUI elements for the Modeling and Monitoring Environment. We satisfy requirement R2 (*choreography instantiation*) with this functionality. The creation of a choreography instance can also be combined with the deployment by retrieving the input parameters prior to deployment and directly instantiating a choreography instance after having deployed all workflow models. This further simplifies the usage of the system for non-IT experts by abstracting from the technical deployment phase.

During instantiation, a unique choreography instance identifier is generated and sent with the initial application message to the workflow engines in a corresponding header, as well as in the application messages between the workflow engines. Additionally, all emitted execution events contain this identifier. This enables the correlation of all events and messages to a particular choreography instance and therefore also its monitoring. The loss of control messages between the middleware and the workflow engines is detected when no corresponding event messages are received after a certain timeout period. The messages can be re-sent assuming idempotent behavior at the workflow engine side.

3.4 ChorSystem Implementation

We have implemented the architecture of the ChorSystem using open source software and open standards. More specifically, we employ BPEL4Chor [9] as the choreography language and BPEL[2] as the 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 Apache ODE[5], and the message routes to compose the middleware functionality are realized with Apache Camel[6]. The interested reader can find more information on the ChorSystem implementation online[7]. Additionally, the ChorSystem Middleware source code has been published on GitHub[8].

4 Evaluation

In the following, we evaluate the performance of the ChorSystem Middleware life cycle operations and illustrate how the operations are triggered from the Modeling and Monitoring Environment. .

4.1 ChorSystem Middleware

The evaluation of our proposal is conducted in an on-premise private cloud environment, on a virtual machine (VM) equipped with 32 GB RAM and 8 virtual CPU cores @3.00 GHz running an Ubuntu 14.04.4 Server 64bit operating system. In the VM, a Docker engine[9] manages a set of Docker containers which host the workflow engines, as well as one container with the ChorSystem Middleware. In order to get comparable results when scaling, the workflow engine containers have an upper memory limit of 1024 MB of memory each, whereas the middleware

[2] `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
[3] `https://eclipse.org/modeling/`
[4] `http://servicemix.apache.org/`
[5] `http://ode.apache.org/`
[6] `http://camel.apache.org/`
[7] `http://www.iaas.uni-stuttgart.de/chorsystem/`
[8] `https://github.com/chorsystem/middleware`
[9] `https://www.docker.com/products/docker-engine`

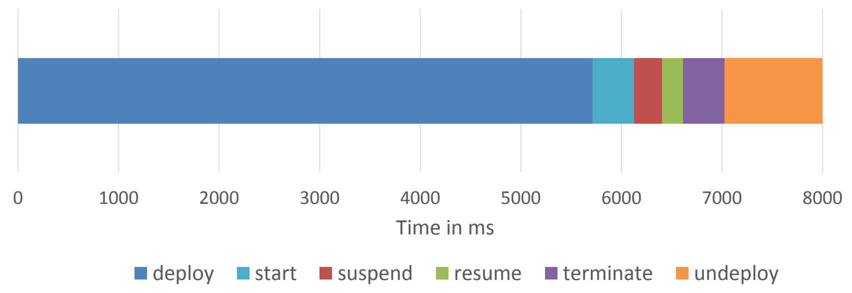


Fig. 6: Processing time share of the life cycle operations for one workflow engine

container is restricted to 4096 MB of memory. Otherwise, the Docker engine would distribute all available memory evenly between each created container. We employ Apache JMeter[10] as load driver and as the means to define, setup, and conduct the evaluation scenarios. JMeter triggers the Docker engine using its REST interface to spin up the necessary number of Docker container instances for each scenario as well as deleting them after measurement to reset the evaluation environment. The JMeter instance is located in a separate VM equipped with 4 GB RAM and 2 virtual CPU cores @3.00 GHz running an Ubuntu 14.04.4 64bit operating system. A choreography model from the ALLOW Ensembles project[11] describing the booking of a trip in a smart city scenario [3] is our choreography model of choice for the evaluation of the life cycle operations. It is more suitable for the evaluation than the motivating example introduced in Fig. 1, because of its significantly larger size. The model is described with the BPEL4Chor language, transformed to executable BPEL and refined thereafter. It consists of 8 participants with a mean size of 24.5 activities each and a mean nesting depth of 4.5 structured activities, for example, BPEL `scope` or `forEach` activities.

When comparing the execution time of the different life cycle operations we observed that the deployment operation in general is the most time consuming one. As shown in Fig. 6, the deploy operation for the trip booking choreography model on one workflow engine node constitutes 70% of the processing time of all life cycle management operations when measured in the above setup. Therefore, in the following we focus on evaluating the execution time of the deploy operation only. The actual execution time of the workflow models implementing the choreography model is excluded completely from this evaluation because their execution is conducted on the workflow engines and is only dependent on the middleware for service lookup, and therefore does not influence the performance of the ChorSystem Middleware itself.

The evaluation is divided into three distinct scenarios. In each scenario, we start with one workflow engine node on which all eight participants are deployed and gradually increase the number of workflow engines first by four, and then in

[10] `http://jmeter.apache.org/`
[11] `http://www.allow-ensembles.eu/`

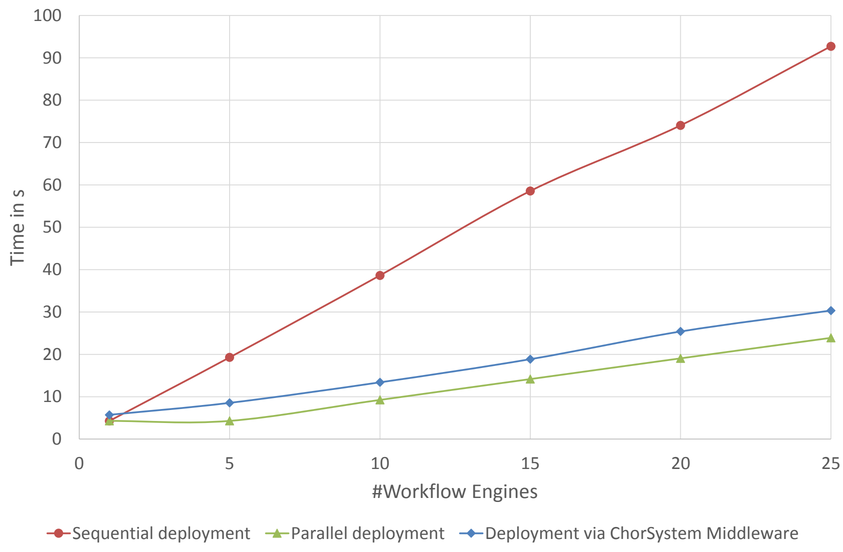


Fig. 7: Choreography deployment execution time

steps of five nodes until we reach 25 nodes in total. The process model deployment is done through the web service interface for life cycle management of the workflow engine. In the *first scenario*, we evaluate the deployment processing time when all eight participant workflow models are directly deployed by JMeter on all running workflow engine nodes in a sequential order, without leveraging the capabilities of the ChorSystem Middleware. In the *second scenario*, each process bundle is sent by JMeter in parallel to the running workflow engine nodes. In the *third scenario*, we employ our ChorSystem Middleware using the information of the choreography deployment descriptor to build a choreography deployment message which is used to distribute the eight choreography participants on the engines.

Figure 7 shows the results of the evaluation[12]. Each data point represents the average of 10 runs. As expected, the parallel scenario is the fastest, because JMeter opens as many threads as workflow engines are available and sends the deployment bundles in parallel. As shown in the figure, it scales linearly with the number of deployment bundles and workflow engines. The sequential scenario also scales linearly but is much slower than the parallel one. The use of the ChorSystem Middleware exhibits a similar performance trend as the parallel scenario. This is due to inherent parallelism (e.g. the implementation of the Recipient List pattern) in the deployment route. The difference to the parallel scenario can be attributed to the overhead induced by the middleware layer for registering all artifacts in the respective registries, and amounts to a deterioration of 45% on average when comparing the two scenarios. However, this overhead seems

[12] The employed choreography model, the workflow models, and the measurement data can be found online: `https://github.com/chorsystem/chorsystem-life-cycle-evaluation`

negligible considering the abilities gained to control the complete choreography life cycle by the ChorSystem.

4.2 ChorSystem Modeling and Monitoring Environment

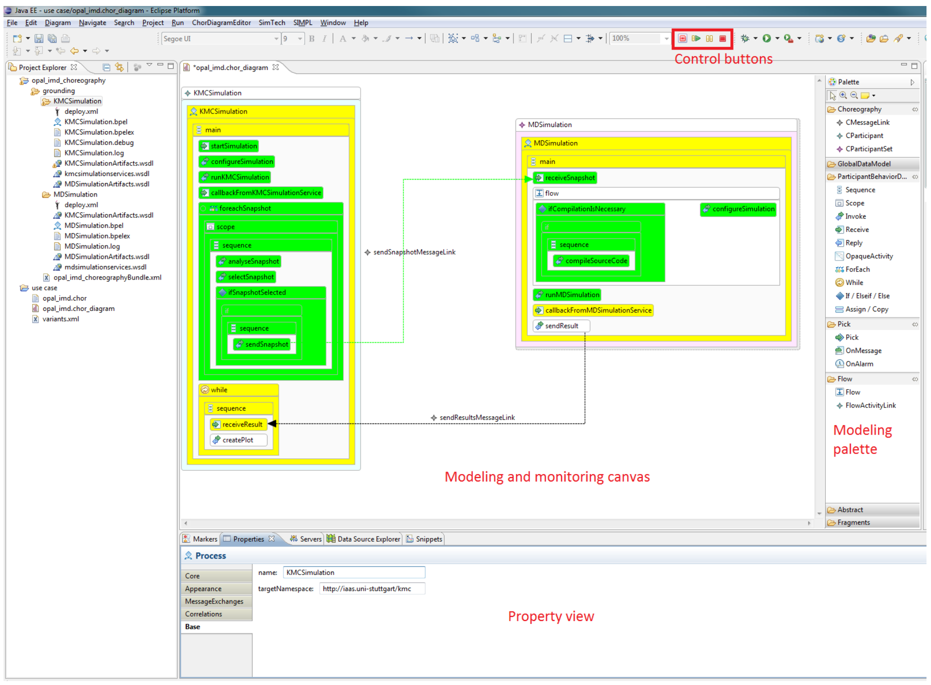
Figure 8 shows a screenshot of our ChorSystem Modeling and Monitoring Environment. The screenshot depicts the choreography introduced in Fig. 1[13]. The user is able to start, pause, resume, and terminate the execution via the corresponding control buttons. If not already conducted in a previous run, the deployment is triggered transparently in the ChorSystem Middleware when pressing the start button. The modeling and monitoring canvas can be used to model choreography logic with the elements provided by the modeling palette. Users with the corresponding access rights can also switch to a more detailed view allowing the refinement of the implementing workflows. During execution time of the workflows, the execution state of each model element present in the choreography model is propagated to the canvas where it is colored accordingly. In Fig. 8, already executed activities are colored green, while currently executing ones are yellow. Correlating instance state with the modeling elements in the depicted manner achieves the desired integration of modeling and monitoring. Authorized users can also switch here to a more detailed view showing the execution state on the workflow level.

5 Related Work

In the following, we compare our work with related ones from literature. In [17], the authors propose an architecture for monitoring cross-organizational executions of choreographies. While the involved organizations execute their workflows autonomously, messages between them are sent through an Enterprise Service Bus and logged reliably. The purpose of logging in this work is run time validation of the choreography execution. Our ChorSystem also uses the benefits of an ESB for message routing as well as logging execution events, however, we do not aim for run time validation but instead want to provide an environment for transparent deployment, user-driven choreography control, flexibility mechanisms as well as integrated modeling and monitoring capabilities. Run time validation is an orthogonal issue that seems promising for integration in future work. The ChorSystem architecture also bears resemblance to the decentralized orchestration, hub-supported architecture style described in [18]. However, we go beyond supporting only the execution of choreographed orchestrations by managing the complete choreography life cycle.

Wang and Pazat [21] use a chemistry-based analogy of tuple spaces to execute distributed participants of a choreography. However, their work does not mention how the deployment of the participant services is handled or how they can be

[13] More screenshots with higher resolutions can be found online: `http://www.iaas.uni-stuttgart.de/chorsystem/`



name: KMCSimulation
targetNamespace: http://iaas.uni-stuttgart/kmc

Fig. 8: Screenshot of the Modeling and Monitoring Environment

actively monitored and controlled. Another nature-inspired approach is presented in [2], where web service compositions are executed by a distributed framework, which is capable of calculating the most suitable service for a task. In [25], an approach for monitoring cross-organizational processes is introduced. The authors utilize complex event processing techniques to achieve their goal. Baouab et al. [5] introduce an approach for the distributed monitoring of choreographies in the context of supply chains. The approach assumes the existence of a hierarchal chain of invocations between the choreography participants and proposes to distribute event messages according to the hierarchy. However, none of these works address our requirements for automated deployment and choreography life cycle management.

The CHOReOS middleware [11] enables the execution of large-scale IoT service choreographies and the probabilistic discovery of services during run time. While this work discusses choreography deployment on an architectural level, it does not provide any details about how the deployment is actually performed. Furthermore, it does not consider user-driven choreography control.

6 Conclusions and Future Work

In this paper, we introduced a system for the management of the complete life cycle from modeling to execution and monitoring of Collaborative, Dynamic &

Complex (CDC) systems which are realized by service choreographies. A ChorSystem was proposed for the management of life cycle operations for choreographies with a message-based ChorSystem Middleware placed between the choreography Modeling and Monitoring Environment and the independent workflow engines executing the choreography participants. The middleware's functionality for choreography deployment/undeployment and control is described and implemented using the Enterprise Integration Patterns. We also proposed the use of a choreography bundle and a choreography deployment descriptor, which captures all information necessary for the transparent deployment of a choreography. The ChorSystem Middleware enables the enactment of life cycle operations based on the globally observed state. We evaluated the performance deviation introduced by the ChorSystem Middleware which shows a processing time in proximity to script-based parallel processing. Thus, we concluded that the performance overhead compared to the parallel scenario is acceptable when the benefits of choreography life cycle management are considered.

Since the ChorSystem so far only considers the initial deployment of a choreography and does not provide support for the addition of new participants or the replacement of failed workflow engines during runtime, we will extend our system to also provide functionality for these scenarios in future. Furthermore, we plan to combine the ChorSystem with the TraDE approach for transparent choreography data management and data flow optimization [10]. The approach decouples choreography data flow from control flow while optimizing the former by analyzing the actual data dependencies between choreography participants and shipping data accordingly. Additionally, we will also integrate our approach with the work presented in [20], where the execution infrastructure for individual workflows is provisioned on demand in a cloud environment. This orthogonal concept can be leveraged to not only deploy the workflow models on existing workflow engines but also provision the workflow engines themselves on demand.

Acknowledgment

This work is funded by the projects FP7 EU-FET 600792 ALLOW Ensembles and the German DFG within the Cluster of Excellence (EXC310/2) Simulation Technology and the DFG Project SitOPT (610872).

References

1. van der Aalst, W., Weske, M.: The p2p approach to interorganizational workflows. In: CAiSE'01, pp. 140–156. Springer (2001)
2. Ahmed, T., Mrissa, M., Srivastava, A.: MagEl: A Magneto-Electric Effect-Inspired Approach for Web Service Composition. In: ICWS'14. pp. 455–462. IEEE (2014)
3. Andrikopoulos, V., Bucchiarone, A., Gómez Sáez, S., Karastoyanova, D., Mezzina, C.A.: Towards Modeling and Execution of Collective Adaptive Systems. In: Proceedings of WESOA'13. pp. 69–81. Springer (2013)

4. Andrikopoulos, V., Gómez Sáez, S., Karastoyanova, D., Weiß, A.: Collaborative, Dynamic & Complex Systems: Modeling, Provision & Execution. In: CLOSER'14. pp. 276–286. SciTePress (2014)
5. Baouab, A., Fdhila, W., Perrin, O., Godart, C.: Towards Decentralized Monitoring of Supply Chains. In: ICWS'12. pp. 600–607. IEEE (2012)
6. Barga, R., Gannon, D.: Scientific versus Business Workflows. In: Workflows for e-Science, pp. 9–16. Springer (2007)
7. Chappell, D.: Enterprise Service Bus. O'Reilly Media, Inc. (2004)
8. Decker, G., Kopp, O., Barros, A.: An Introduction to Service Choreographies. Inf. Technology 50(2), 122–127 (2008)
9. Decker, G., Kopp, O., Leymann, F., Weske, M.: Interacting services: from specification to execution. Data & Knowledge Engineering 68(10), 946–972 (2009)
10. Hahn, M., Karastoyanova, D., Leymann, F.: Data-Aware Service Choreographies through Transparent Data Exchange. In: ICWE'16. pp. 357–364. Springer (2016)
11. Hamida, A. B. et. al: Integrated CHOReOS middleware-Enabling large-scale, QoS-aware adaptive choreographies (2013)
12. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2004)
13. Knuplesch, D., Reichert, M., Pryss, R., Fdhila, W., Rinderle-Ma, S.: Ensuring compliance of distributed and collaborative workflows. In: Collaboratecom'13. pp. 133–142. IEEE (2013)
14. Kopp, O., van Lessen, T., Nitzsche, J.: The Need for a Choreography-aware Service Bus. In: YR-SOC 2008. pp. 28–34 (2008)
15. Molnar, D., Mukherjee, R., Choudhury, A., Mora, A., Binkele, P., Selzer, M., Nestler, B., Schmauder, S.: Multiscale simulations on the coarsening of cu-rich precipitates in a-fe using kinetic monte carlo, molecular dynamics and phase-field simulations. Acta Materialia 60(20), 6961–6971 (2012)
16. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: A research roadmap. Int. J. of Coop. Inf. Systems 17(02), 223–255 (2008)
17. von Riegen, M., Ritter, N.: Reliable Monitoring for Runtime Validation of Choreographies. In: ICIW'09. pp. 310–315. IEEE (2009)
18. Schroth, C., Janner, T., Hoyer, V.: Strategies for Cross-Organizational Service Composition. In: Int. MCETECH Conf. on e-Technologies. pp. 93–103 (2008)
19. Sonntag, M., Karastoyanova, D.: Model-as-you-go: An Approach for an Advanced Infrastructure for Scientific Workflows. Grid Computing 11(3), 553–583 (2013)
20. Vukojevic-Haupt, K., Karastoyanova, D., Leymann, F.: On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows. In: SOCA'13. pp. 91–98. IEEE (2013)
21. Wang, C., Pazat, J.L.: A Chemistry-Inspired Middleware for Self-Adaptive Service Orchestration and Choreography. In: CCGrid'13. pp. 426–433 (2013)
22. Weiß, A., Andrikopoulos, V., Hahn, M., Karastoyanova, D.: Rewinding and Repeating Scientific Choreographies. In: OTM'15. pp. 337–347. Springer (2015)
23. Weiß, A., Karastoyanova, D.: A Life Cycle for Coupled Multi-Scale, Multi-Field Experiments Realized through Choreographies. In: EDOC'14. pp. 234–241. IEEE (2014)
24. Weiß, A., Karastoyanova, D.: Enabling coupled multi-scale, multi-field experiments through choreographies of data-driven scientific simulations. Computing 98(4), 439–467 (2016)
25. Wetzstein, B., Karastoyanova, D., Kopp, O., Leymann, F., Zwink, D.: Cross-Organizational Process Monitoring based on Service Choreographies. In: SAC'10. pp. 2485–2490. ACM (2010)